目　录

前　　言

随着 Linux 操作系统的发展，特别是 Linux 2.6 内核的迅速发展，嵌入式 Linux 在嵌入式领域的应用越来越广泛。Linux 具备源码开放、内核稳定高效、软件丰富等优势，而且还具备支持广泛的处理器结构和硬件平台、可定制性好、可靠性高等特点。据 IDC 的报告显示，嵌入式 Linux 在未来两年将占嵌入式操作系统市场份额的 50%，约 3.5 亿美元，由此产生的应用市场前景更是不可估量。

正是由于市场的需求，嵌入式领域也需要大量的嵌入式 Linux 开发者。目前国内 Linux 程序员的素质和数量还不能满足企业的需要。

编写目的

大学计算机相关专业课程都已经包含计算机组成原理、计算机编程语言、计算机体系结构、计算机操作系统，甚至还包括电子技术和半导体技术。尽管已经具备这些嵌入式 Linux 系统的基础知识，但是多数大学毕业生不清楚到底该如何开发嵌入式 Linux 系统。

编写本书的目的就是阐述嵌入式 Linux 系统的各组成部分，从概念上和实践上说明嵌入式 Linux 系统开发的基本过程。这本书可以帮助具备计算机基础知识的开发者迅速进入嵌入式系统开发领域。

希望本书能够帮助读者更好地理解嵌入式 Linux 系统，并且参与到嵌入式 Linux 系统开发中来。

主要内容

本书以嵌入式 Linux 系统开发流程为主线，剖析了嵌入式 Linux 系统构建的各个环节。从嵌入式系统基础知识和 Linux 编程技术讲起，接下来说明了建立嵌入式 Linux 交叉开发环境，然后分析了嵌入式 Linux 系统的引导程序、内核和文件系统三大组成部分，最后介绍了嵌入式 Linux 系统集成和部署的方法。

第 1 章介绍了嵌入式系统和嵌入式操作系统的概况，讲述了嵌入式 Linux 发展历史和开发环境，概括说明了嵌入式 Linux 系统开发的特点。

第 2 章描述了 ARM 体系结构和 ARM Linux 的发展，介绍了几种应用 Linux 的典型 ARM 处理器和开发板。

第 3 章介绍了 Linux 编程常用的工具，Makefile 语法规则，还有 binutils、gcc 和 gdb 等工具的用法。

第 4 章介绍了嵌入式交叉开发环境的概念和配置，说明了应用程序交叉开发和调试的基本方法。

第 5 章介绍了编译生成 GNU 工具链的基本步骤。

第 6 章介绍了 Bootloader 的类型的特点，详细分析了 U-Boot 的使用、编译和移植。

第 7 章介绍了 Linux 2.6 内核的特点和 Kbuild 管理方式，说明了内核基本的配置选项的用法。

第 8 章以 ARM 平台为例介绍了内核移植的基本方法，并且详细分析了 Linux 内核启动过程。

第 9 章介绍了各种 Linux 内核调试方法，为内核移植提供了有效的调试手段。

第 10 章介绍了 Linux 根文件系统的组织结构，并且分析了 init 进程调用文件系统脚本初始化的过程。

第 11 章介绍了嵌入式 Linux 系统常用的开源软件，包括系统工具、图形库、网络和串口应用程序等。

第 12 章介绍了系统集成测试需要的各种工具，主要包括系统跟踪、性能测试和内存测试 3 个方面。

第 13 章介绍了 Linux 系统部署的基本方法，分析了文件系统和存储介质的特点。

第 14 章介绍了以 S3C2410 处理器的 GPS 手持设备开发过程为例，介绍了嵌入式 Linux 系统软硬件的设计与开发。

本书可作为高等院校电子类、电气类、控制类等专业高年级本科生、研究生学习嵌入式 Linux 的教材，也可供广大希望转入嵌入式领域科研和工程技术人员参考使用，还可作为广大嵌入式 Linux 就业培训班的教材和教辅材料。

阅读建议

根据本书的指导，可以自己动手构建嵌入式 Linux 开发环境和嵌入式 Linux 系统。这对于深刻理解和掌握嵌入式 Linux 开发是非常重要的。

嵌入式的开发与具体的硬件环境紧密相关，本书的内容以常见的 ARM 9 S3C2410 平台为例来讲解。对于其他硬件平台可以触类旁通，通过分析具体的源代码学习。

感谢

感谢张小全的密切合作，他的努力使得本书得以及时完稿，他撰写了第 2 章、第 3 章、第 10 章、第 11 章。

感谢开放源码软件和开放文档的作者们。

感谢华清远见的季久峰老师。

感谢麦克泰公司和我的同事们，特别是支持我的龙中花和易松华。

另外还要感谢以下人员的支持：孙天泽、袁文菊、田彦、周明、黄昕、史宜彬、张秀丽、谭翠君等。

相关内容

本书内容来自北京华清远见科技信息有限公司（http://www.farsight.com.cn）的培训课程资料，有关本书的相关资料和嵌入式Linux更多的资料、公开课视频，请参见http://www.farsight.com.cn/download/。

由于时间仓促，加之水平有限，书中的不足之处在所难免，敬请读者批评指正。本书责任编辑的联系方法是quyanlian2@ptpress.com.cn，欢迎来信交流。

编者

2006年6月

“黑色经典”系列之《嵌入式 Linux 系统开发技术详解——基于 ARM》

第 1 章 概述

本章目标

本章主要介绍嵌入式系统和嵌入式操作系统的概况，讲述嵌入式 Linux 的发展历史和开发环境，概括说明嵌入式 Linux 系统开发的特点。读完本章内容，可以对嵌入式 Linux 系统有整体的认识，了解嵌入式 Linux 开发的要点。

- 嵌入式系统定义
- 嵌入式操作系统介绍
- 嵌入式 Linux 操作系统
- 嵌入式 Linux 开发环境
- 嵌入式 Linux 系统开发要点

1.1 嵌入式系统

嵌入式系统是以应用为中心，以计算机技术为基础，软硬件可裁剪，适用于应用系统，对功能、可靠性、成本、体积、功耗等方面有特殊要求的专用计算机系统。

嵌入式系统与通用计算机系统的本质区别在于系统应用不同，嵌入式系统是将一个计算机系统嵌入到对象系统中。这个对象可能是庞大的机器，也可能是小巧的手持设备，用户并不关心这个计算机系统的存在。

嵌入式系统一般包含嵌入式微处理器、外围硬件设备、嵌入式操作系统和应用程序 4 个部分。嵌入式领域已经有丰富的软硬件资源可以选择，涵盖了通信、网络、工业控制、消费电子、汽车电子等各种行业。

嵌入式计算机系统与通用计算机系统相比具有以下特点。

（1）嵌入式系统是面向特定系统应用的。嵌入式处理器大多数是专门为特定应用设计的，具有低功耗、体积小、集成度高等特点，一般是包含各种外围设备接口的片上系统。

（2）嵌入式系统涉及计算机技术、微电子技术、电子技术、通信和软件等各行各业。它是一个技术密集、资金密集、高度分散、不断创新的知识集成系统。

（3）嵌入式系统的硬件和软件都必须具备高度可定制性。只有这样才能适用嵌入式系统应用的需要，在产品价格性能等方面具备竞争力。

（4）嵌入式系统的生命周期相当长。当嵌入式系统应用到产品以后，还可以进行软件升级，它的生命周期与产品的生命周期几乎一样长。

（5）嵌入式系统不具备本地系统开发能力，通常需要有一套专门的开发工具和环境。

在计算机后 PC 技术时代，嵌入式系统将拥有最大的市场。计算机和网络已经全面渗透到日常生活的每一个角落。各种各样的新型嵌入式系统设备在应用数量上已经远远超过通用计算机，任何一个普通人可能拥有从大到小的各种使用嵌入式技术的电子产品，小到 MP3、PDA 等微型数字化产品，大到网络家电、智能家电、车载电子设备。而在工业和服务领域中，使用嵌入式技术的数字机床、智能工具、工业机器人、服务机器人也将逐渐改变传统的工业和服务方式。

美国著名的未来学家尼葛洛庞帝在 1999 年访华时曾预言，4～5 年后嵌入式系统将是继 PC 和 Internet 之后最伟大的发明。这个预言已经成为现实，现在的嵌入式系统正处于高速发展阶段。

1.2 嵌入式操作系统

嵌入式操作系统的一个重要特性是实时性。所谓实时性，就是在确定的时间范围内响应某个事件的特性。操作系统的实时性在某些领域是至关重要的，比如工业控制、航空航天等领域。想像飞机正在空中飞行，如果嵌入式系统不能及时响应飞行员的控制指令，那么极有可能导致空难事故。有些嵌入式系统应用并不需要绝对的实时性，比如 PDA 播放音乐，个别音频数据丢失并不影响效果。这可以使用软实时的概念来衡量。

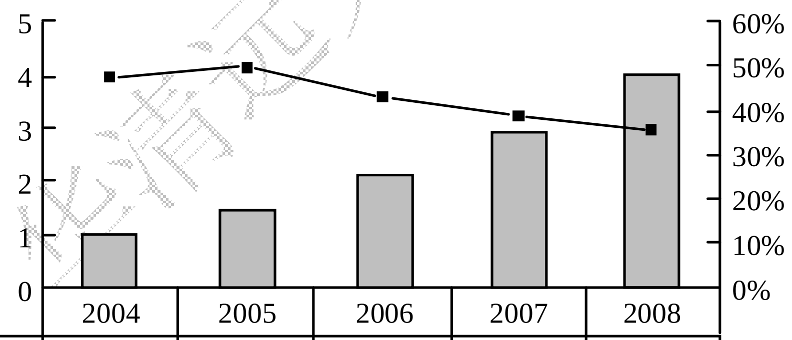
据调查，目前全世界的嵌入式操作系统已经有两百多种。从 20 世纪 80 年代开始，出现了一些商用嵌入式操作系统，它们大部分都是为专有系统而开发的。随着嵌入式领域的发展，各种各样嵌入式操作系统相继问世。有许多商业的嵌入式操作系统，也有大量开放源码的嵌入式操作系统。其中著名的嵌入式操作系统有：μC/OS、VxWorks、Neculeus、Linux 和 Windows CE 等。下面介绍一些主流的嵌入式操作系统。

（1）Linux

在所有的操作系统中，Linux 是一个发展最快、应用最为广泛的操作系统。Linux 本身的种种特性使其成为嵌入式开发中的首选。在进入市场的头两年中，嵌入式 Linux 设计通过广泛应用获得了巨大的成功。随着嵌入式 Linux 的的成熟，提供更小的尺寸和更多类型的处理器支持，并从早期的试用阶段迈进到嵌入式的主流，它抓住了电子消费类设备的开发者们的想像力。图 1.1 所示是业内人士对国内 Linux 软件市场的预测。

根据 IDC 的报告，Linux 已经成为全球第二大操作系统。预计在服务器市场上，Linux 在未来几年内将以每年 25%的速度增长，中国的 Linux 市场更是保持 40%左右的增长速度。而在 Linux 操作系统方面，IDC 对中国在 2001～2006 年的市场预测发现，其市场占有率从 2001 年的 4.47%平稳地上升到 2006 年的 26.77%。

嵌入式 Linux 版本还有多种变体。例如：RTLinux 通过改造内核实现了实时的 Linux；



	2004	2005	2006	2007	2008
销售额（亿元）	0.99	1.48	2.11	2.93	3.96
增长率（%）	47.3%	49.4%	42.6%	38.5%	35.1%

图 1.1　2004～2008 年国内 Linux 软件市场总量预测

RTAI、Kurt 和 Linux/RK 也提供了实时能力；还有μCLinux 去掉了 Linux 的 MMU（内存管理单元），能够支持没有 MMU 的处理器等。

（2）μC/OS

μC/OS 是一个典型的实时操作系统。该系统从 1992 年开始发展，目前流行的是第 2 个版本，即μC/OS II。它的特点是：公开源代码，代码结构清晰，注释详尽，组织有条理，可移植性好；可裁剪，可固化；抢占式内核，最多可以管理 60 个任务。自从清华大学邵贝贝教授将 Jean J. Labrosse 的《μC/OS-Ⅱ：the Real Time Kernel》翻译后，在国内掀起μC/OS II 的热潮，特别是在教育研究领域。该系统短小精悍，是研究和学习实时操作系统的首选。

（3）Windows CE

Windows CE 是微软的产品，它是从整体上为有限资源的平台设计的多线程、完整优先权、多任务的操作系统。Windows CE 采用模块化设计，并允许它对于从掌上电脑到专用的工控电子设备进行定制。操作系统的基本内核需要至少 200KB 的 ROM。从 SEGA 的 DreamCast 游戏机到现在大部分的高价掌上电脑都采用了 Windows CE。

随着嵌入式操作系统领域日益激烈的竞争，微软不得不应付来自 Linux 等免费系统的冲击。微软在 Windows CE.Net 4.2 版中，将增加一项授权价仅 3 美元的精简版本 WinCE.Net Core。WinCE.Net Core 具有基本的功能，包括实时 OS 核心（Real Time OS Kernel）、档案系统；IPv4、IPv6、WLAN、蓝牙等联网功能；Windows Media Codec；.Net 开发框架以及 SQL Server.ce。微软推出低价版本 WinCE.Net，主要是看好语音电话、WLAN 的无线桥接器和个性化视听设备的成长潜力。

（4）VxWorks

VxWorks 是 WindRiver 公司专门为实时嵌入式系统设计开发的操作系统软件，为程序员提供了高效的实时任务调度、中断管理，实时的系统资源以及实时的任务间通信。应用程序员可以将尽可能多的精力放在应用程序本身，而不必再去关心系统资源的管理。该系统主要应用在单板机、数据网络（以太网交换机、路由器）和通信方面等多方面。其核心功能主要有以下几个。

- 微内核 wind
- 任务间通信机制
- 网络支持
- 文件系统和 I/O 管理
- POSIX 标准实时扩展
- C++以及其他标准支持

这些核心功能可以与 WindRiver 系统的其他附件和 Tornado 合作伙伴的产品结合在一起使用。谁都不能否认这是一个非常优秀的实时系统，但其昂贵的价格使不少厂商望而却步。

（5）QNX

这也是一款实时操作系统，由加拿大 QNX 软件系统有限公司开发。广泛应用于自动化、控制、机器人科学、电信、数据通信、航空航天、计算机网络系统、医疗仪器设备、交通运输、安全防卫系统、POS 机、零售机等任务关键型应用领域。20 世纪 90 年代后期，QNX 系统在高速增长的因特网终端设备、信息家电及掌上电脑等领域也得到了广泛应用。

QNX 的体系结构决定了它具有非常好的伸缩性，用户可以把应用程序代码和 QNX 内核直接编译在一起，使之为简单的嵌入式应用生成一个单一的多线程映像。它也是世界上第一个遵循 POSIX1003.1 标准从零设计的微内核，因此具有非常好的可移植性。

嵌入式操作系统的选择是前期设计过程的一项重要工作，这将影响到工程后期的发布以及软件的维护。不管选用什么样的系统，都应该考虑操作系统对硬件的支持，如果选择的系统不支持将来要使用的硬件平台，那这个系统是不合适的；其次要考虑的是开发调试用的工具，特别是对于开销敏感和技术水平不强的企业来说，开发工具往往在开发过程中起决定性作用；第三要考虑的问题是该系统能否满足应用需求。如果一个操作系统提供出来的 API 很

少，那么无论这个系统有多么稳定，应用层很难进行二次开发，这显然也不是开发人员希望看到的。由此可见，选择一款既能满足应用需求，性价比又可达到最佳的实时操作系统，对开发工作的顺利开展意义非常重大。

1.3 嵌入式 Linux 历史

所谓嵌入式 Linux，是指 Linux 在嵌入式系统中应用，而不是什么嵌入式功能。实际上，嵌入式 Linux 和 Linux 是同一件事。

我们了解一下 Linux 的发展历史。

Linux 起源于 1991 年，由芬兰的 Linus Torvalds 开发，随后按照 GPL 原则发布。

Linux 1.0 正式发行于 1994 年 3 月，仅支持 386 的单处理器系统。

Linux 1.2 发行于 1995 年 3 月，它是第一个包含多平台（Alpha，Sparc，Mips 等）支持的官方版本。

Linux 2.0 发行于 1996 年 6 月，包含很多新的平台支持。最重要的是，它是第一个支持 SMP（对称多处理器）体系的内核版本。

Linux 2.2 于 1999 年 1 月发布，它带来了 SMP 系统上性能的极大提升，同时支持更多的硬件。

Linux 2.4 于 2001 年 1 月发布，它进一步提升了 SMP 系统的扩展性，同时它也集成了很多用于支持桌面系统的特性：USB，PC 卡（PCMCIA）的支持，内置的即插即用，等等。

Linux 2.6 于 2003 年 12 月发布，它的多种内核机制都有了重大改进，无论对大系统还是小系统（PDA 等）的支持都有很大提高。

最新的 Linux 内核版本可以从官方站点获取。

```
http://www.kernel.org
```

Linux 是一种类 UNIX 操作系统。从绝对意义上讲，Linux 是 Linus Torvalds 维护的内核。现在的 Linux 操作系统已经包括内核和大量应用程序，这些软件大部分来源于 GNU 软件工程。因此，Linux 又叫作 GNU/Linux。

目前 Linux 操作系统的发行版已经有很多，例如：Redhat Linux、Suse Linux、Turbo Linux 等台式机或者服务器版本，还有各种嵌入式 Linux 版本。不同的 Linux 版本之间总会有些差异。鉴于 UNIX 技术历史的教训，LSB（Linux Standard Base）为 Linux 系统制定了规范。LSB 规范定义了几种模块，并且为应用程序定义系统接口和基本配置，为大量的应用程序提供了统一的行业标准。从以下站点可以获取 LSB 的文档。

```
http://www.linuxbase.org
```

ELC（Embedded Linux Consortium，嵌入式 Linux 联盟）是一个非营利性的、中立的行业协会，它的目标是在嵌入式应用和设备计算市场做 Linux 的改进、推广和标准化工作。联盟成员贡献会费并且参与管理、推广、实现和平台规范工作组的维护，谋求不断增长的市场机遇。ELC 成员为了 API 的互用性积极推广一套平台标准，消除分割并且发布更加具有竞争

力的商业方案。

http://www.embedded-linux.org

OSDL（Open Source Development Labs）支持围绕 Linux 开发和指导的各种活动。它为 OSDL 协会免费提供硬件资源。OSDL 发起了电信 Linux（Carrier Grade Linux）和数据中心 Linux（Data Center Linux）工作组。这些工作组包含 OSDL 成员和有兴趣的个人，他们致力于创建特点列表和规范，并且参与开源工程为电信和数据中心进一步开发 Linux。OSDL 还积极参与内核测试，提供了开放的测试环境（Scalable Test Platform），并且贡献给开发状态的内核测试。

http://www.osdl.org

CELF（Consumer Electronics Linux Forum，消费电子 Linux 论坛）是加州的一个非盈利性公司，它致力于把 Linux 改进成消费电子设备的开放平台。

http://www.celinuxforum.org

越来越多的个人、社团和公司已经和正在参与 Linux 社区的工作，他们为 Linux 系统开发、测试以及应用做了大量贡献。这使得嵌入式 Linux 系统成为标准化的操作系统，功能日趋完善，应用更加广泛。

1.4 嵌入式 Linux 开发环境

通用计算机可以直接安装发行版的 Linux 操作系统，使用编辑器、编译器等工具为本机开发软件，甚至可以完成整个 Linux 系统的升级。

嵌入式系统的硬件一般有很大的局限性，或者处理器频率很低，或者存储空间很小，或者没有键盘、鼠标设备。这样的硬件平台无法胜任（或者不便于）庞大的 Linux 系统开发任务。因此，开发者提出了交叉开发环境模型。

交叉开发环境是由开发主机和目标板两套计算机系统构成的。目标板 Linux 软件是在开发主机上编辑、编译，然后加载到目标板上运行的。为了方便 Linux 内核和应用程序软件的开发，还要借助各种连接手段。第 4 章将详细介绍如何建立交叉开发环境。第 5 章将详细介绍交叉编译工具链的建立。

Linux 是开放源码的软件工程，它的大量应用程序也来源于 GNU 软件工程。因此，完全可以自己动手制作一套完整的嵌入式 Linux 系统和开发工具。但是，庞大的系统软件开发和测试将花费大量人工时，无法预料的 BUG 可能严重阻滞项目进度。

大量的开源软件和商业的 Linux 软件共同出现在 Linux 操作系统上，半导体公司、Linux 操作系统公司、第三方软件公司等已经形成庞大的 Linux 生态系统。任何一家公司都不可能对 Linux 系统做全面的维护和技术支持。

嵌入式 Linux 系统的开发工具绝大多数是命令行方式的，这使得学习 Linux 开发比 Windows 开发难度更大。商业公司在嵌入式 Linux 产品开发的时候，希望有更方便、更快捷

的开发工具可以使用。因此，嵌入式 Linux 集成开发环境具有市场需求。

目前，Eclipse 已经成为集成开发环境的标准平台。Eclipse 是开放的、跨平台的、高度可配置的集成开发环境，它已经被众多嵌入式操作系统厂商定制成自己的集成开发环境。例如：MontaVista 公司的 DevRocket、TimeSys 公司的 TimeStorm、Wind River 公司的 Workbench。

MontaVista DevRocket 集成开发环境如图 1.2 所示。

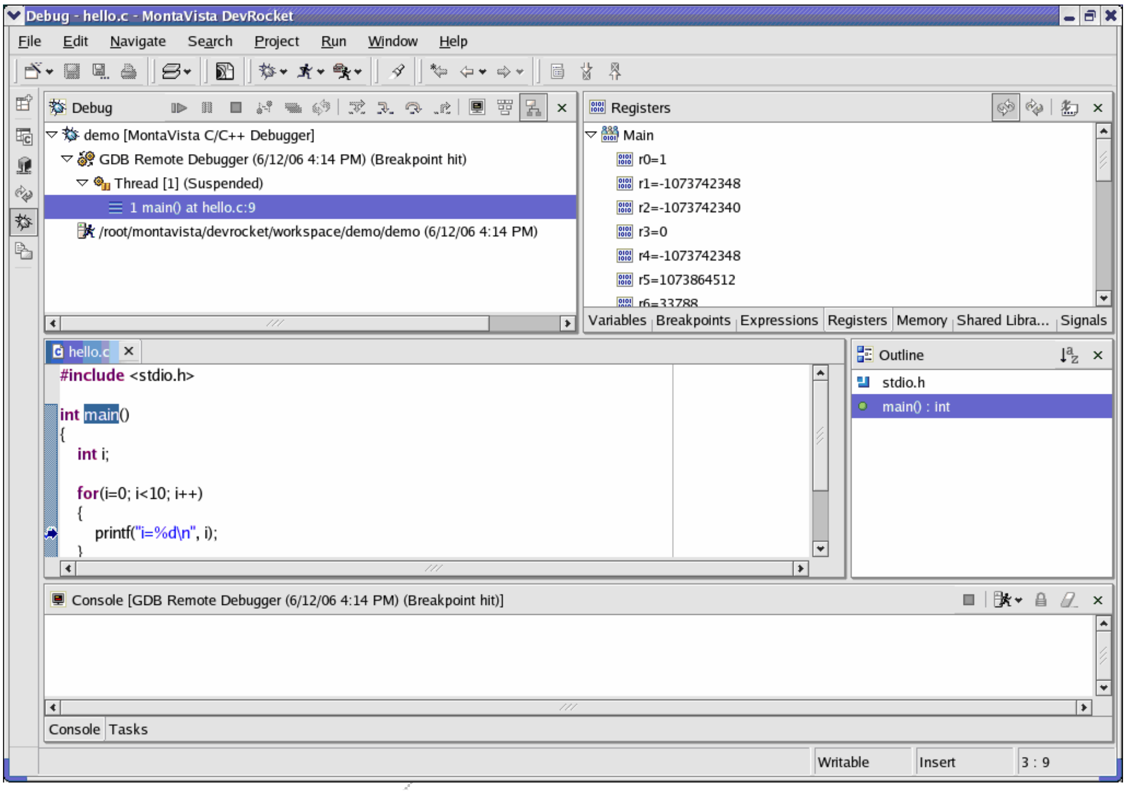


图 1.2　MontaVista DevRocket 集成开发环境

这些集成开发环境不但能够支持应用程序开发和调试，而且专门提供了内核、文件系统的工程。另外可以集成各种测试工具和版本控制等功能，大大方便了嵌入式 Linux 开发。

1.5　嵌入式 Linux 系统开发要点

嵌入式 Linux 开发就是构建一个 Linux 系统，这需要熟悉 Linux 系统组成部分，熟悉 Linux 开发工具，还要熟悉 Linux 编程。

嵌入式 Linux 系统包含 Bootloader（引导程序）、内核和文件系统 3 部分。对于嵌入式 Linux 系统来说，这 3 个部分是必不可少的。本书将详细分析这 3 个部分的相关软件开发。

总之，在启动一个嵌入式 Linux 项目之前，必须仔细考虑下面要点。

（1）选择嵌入式 Linux 发行版

商业的 Linux 发行版是作为产品开发维护的，经过严格的测试验证，并且可以得到厂家

的技术支持。它为开发者提供了可靠的软件和完整的开发工具包。

（2）熟悉开发环境和工具

交叉开发环境是嵌入式 Linux 开发的基本模型。Linux 环境配置、GNU 工具链、测试工具甚至集成开发环境都是开发嵌入式 Linux 开发的利器。

（3）熟悉 Linux 内核

因为嵌入式 Linux 开发一般需要重新定制 Linux 内核，所以熟悉内核配置、编译和移植也很重要。

（4）熟悉目标板引导方式

开发板的 Bootloader 负责硬件平台的最基本的初始化，并且具备引导 Linux 内核启动的功能。由于硬件平台是专门定制的，一般需要修改编译 Bootloader。

（5）熟悉 Linux 根文件系统

Linux 离不开文件系统，程序和文件都存放在文件系统中。系统启动必需的程序和文件都必须放在根文件系统中。Linux 内核命令行参数可以指定要挂接的根文件系统。

（6）理解 Linux 内存模型

Linux 是保护模式的操作系统。内核和应用程序分别运行在完全分离的虚拟地址空间，物理地址必须映射到虚拟地址才能访问。只有理解 Linux 内存模型，才能最大程度地优化系统性能。

（7）理解 Linux 调度机制和进程线程编程

Linux 调度机制影响到任务的实时性，理解调度机制可以更好地运用任务优先级。进程和线程编程则是应用程序开发所必需的。

“黑色经典”系列之《嵌入式 Linux 系统开发技术详解——基于 ARM》

第 2 章　ARM 处理器

本章目标

本章描述了 ARM 体系结构和 ARM Linux 的发展，介绍了几种应用 Linux 的典型 ARM 处理器和开发板。本章可以使读者了解嵌入式 Linux 系统硬件平台的基础知识。

- ARM 体系结构
- 典型的 ARM 处理器
- S3C2410 开发板介绍

2.1 ARM 处理器简介

ARM（Advanced RISC Machines），既可以认为是一个公司的名字，也可以认为是对一类微处理器的通称，还可以认为是一种技术的名字。ARM 处理器是一种低功耗高性能的 32 位 RISC 处理器，ARM 处理器是一个综合体，ARM 公司自身并不制造微处理器，而是由 ARM 的合作伙伴来制造，作为 SOC（System On Chip）的典型应用，目前，基于 ARM 的处理器以其高速度、低功耗等诸多优异的性能而得到非常广泛的应用。

采用 RISC 架构的 ARM 微处理器一般具有如下特点。

- 体积小、低功耗、低成本、高性能。
- 支持 Thumb（16 位）/ARM（32 位）双指令集，能很好地兼容 8 位/16 位器件。

ARM 微处理器支持 2 种指令集：ARM 指令集和 Thumb 指令集。其中，ARM 指令为 32 位的长度，Thumb 指令为 16 位长度。Thumb 指令集为 ARM 指令集的功能子集，但与等价的 ARM 代码相比较，可节省 30%～40%以上的存储空间，同时具备 32 位代码的所有优点。

- 大量使用寄存器，指令执行速度更快。

ARM 处理器共有 37 个寄存器，被分为若干个组（BANK），如下。

— 31 个通用寄存器，包括程序计数器（PC 指针），均为 32 位的寄存器。

— 6 个状态寄存器，用以标识 CPU 的工作状态及程序的运行状态，均为 32 位。

概括地讲，ARM 体系结构中各寄存器的使用方式可以归纳如表 2.1 所示。

表 2.1 ARM 寄存器使用方式

寄 存 器	使 用 方 式
程序计数器 PC（r15）	所有运行状态都可以使用
通用寄存器 r0～r7	所有运行状态都可以使用
通用寄存器 r8～r12	除去快速中断以外的状态都可以使用
当前程序状态寄存器 CPSR	所有运行状态都可以使用
保存程序状态寄存器 SPSR	除去用户状态以外的 6 种运行状态，分别都有自己的 SPSR
堆栈指针 SP（r13）和链接寄存器 lr（r14）	所有的运行状态都有自己的 SP 和 lr

- 大多数数据操作都在寄存器中完成。
- 寻址方式灵活简单，执行效率高。
- 指令长度固定。

为了保证 ARM 处理器具有高性能的同时，进一步减少芯片的体积和功耗，ARM 处理器采用了以下一些比较特别的技术。

- 所有的指令都可根据前面的执行结果决定是否被执行，从而提高指令的执行效率。

- 可用加载/存储指令批量传输数据，以提高数据的传输效率。
- 可在一条数据处理指令中同时完成逻辑处理和移位处理。
- 在循环处理中使用地址的自动增减来提高运行效率。

ARM 微处理器有以下 7 种运行模式。

- 用户模式（usr）：ARM 处理器正常的程序执行状态。
- 快速中断模式（fiq）：用于高速数据传输或通道处理。
- 外部中断模式（irq）：用于通常的中断处理。
- 管理模式（svc）：操作系统使用的保护模式。
- 数据访问终止模式（abt）：当数据或指令预取终止时进入该模式，可用于虚拟存储及存储保护。
- 系统模式（sys）：运行具有特权的操作系统任务。
- 未定义指令中止模式（und）：当未定义的指令执行时进入该模式，可用于支持硬件协处理器的软件仿真。

ARM 微处理器的运行模式可以通过软件改变，也可以通过外部中断或异常处理改变。大多数的应用程序运行在用户模式下，当处理器运行在用户模式下时，某些被保护的系统资源是不能被访问的。除用户模式以外，其余的所有 6 种模式称之为非用户模式，或特权模式（Privileged Modes）；其中除去用户模式和系统模式以外的 5 种又称为异常模式（Exception Modes），常用于处理中断或异常，以及需要访问受保护的系统资源等情况。

2.1.1　ARM 公司简介

1991 年 ARM 公司（Advanced RISC Machine Limited）成立于英国剑桥，最早由 Arcon、Apple 和 VLSI 合资成立，主要出售芯片设计技术的授权，在 1985 年 4 月 26 日，第一个 ARM 原型在英国剑桥的 Acorn 计算机有限公司诞生（在美国 VLSI 公司制造）。目前，ARM 架构处理器已在高性能、低功耗、低成本的嵌入式应用领域中占据了领先地位。

ARM 公司最初只有 12 人，经过十多年的发展，ARM 公司已拥有近千名员工，在许多国家都设立了分公司，包括 ARM 公司在中国上海的分公司。目前，采用 ARM 技术知识产权（IP）核的微处理器，即我们通常所说的 ARM 微处理器，已遍及工业控制、消费类电子产品、通信系统、网络系统、无线系统等各类产品市场，基于 ARM 技术的微处理器应用约占据了 32 位 RISC 微处理器 80%以上的市场份额，其中，在手机市场，ARM 占有绝对的垄断地位。可以说，ARM 技术正在逐步渗入到人们生活中的各个方面，而且随着 32 位 CPU 价格的不断下降和开发环境的不断成熟，ARM 技术会应用得越来越广泛。

ARM 公司是专门从事基于 RISC 技术芯片设计开发的公司，作为嵌入式 RISC 处理器的知识产权 IP 供应商，公司本身并不直接从事芯片生产，而是靠转让设计许可由合作公司生产各具特色的芯片，世界各大半导体生产商从 ARM 公司购买其设计的 ARM 微处理器核，根据各自不同的应用领域，加入适当的外围电路，从而形成自己的 ARM 微处理器芯片进入市场，利用这种合伙关系，ARM 很快成为许多全球性 RISC 标准的缔造者。目前，全世界有几十家大的半导体公司都使用 ARM 公司的授权，其中包括 Intel、IBM、Samsung、LG 半导体、NEC、SONY、PHILIP 等公司，这也使得 ARM 技术获得更多的第三方工具、制造、软件的支持，又使整个系统成本降低，使产品更容易进入市场并被消费者所接受，更具有竞争力。

2.1.2 ARM 处理器体系结构

处理器的体系结构定义了指令集（ISA）和基于这一体系结构下处理器的程序员模型。ARM 体系结构为嵌入系统发展商提供了很高的系统性能，同时保持了优异的功耗和面积效率，每一次 ARM 体系结构的重大修改，都会添加一些非常关键的技术。

目前，ARM 体系结构共定义了 6 个版本，从版本 1 到版本 6，ARM 体系的指令集功能不断扩大，不同系列的 ARM 处理器，性能差别很大，应用范围和对象也不尽相同，但是，如果是相同的 ARM 体系结构，那么基于它们的应用软件是兼容的。

1．V1 结构（版本 1）

V1 版本的 ARM 处理器并没有实现商品化，采用的地址空间是 26 位，寻址空间是 64MB，在目前的版本中已不再使用这种结构。

2．V2 结构

与 V1 结构的 ARM 处理器相比，V2 结构的 ARM 处理器的指令结构要有所完善，比如增加了乘法指令并且支持协处理器指令，在该版本的处理器仍然是 26 位的地址空间。

3．V3 结构

从 V3 结构开始，ARM 处理器的体系结构有了很大的改变，实现了 32 位的地址空间，指令结构相对前面的两种结构也所完善。

4．V4 结构

V4 结构的 ARM 处理器增加了半字指令的读取和写入操作，增加了处理器系统模式，并且有了 T 变种-V4T，在 Thumb 状态下所支持的是 16 位的 Thumb 指令集。属于 V4T（支持 Thumb 指令）体系结构的处理器（核）有 ARM7TDMI，ARM7TDMI-S（ARM7TDMI 可综合版本），ARM710T（ARM7TDMI 核的处理器），ARM720T（ARM7TDMI 核的处理器），ARM740T（ARM7TDMI 核的处理器），ARM9TDMI，ARM910T（ARM9TDMI 核的处理器），ARM920T（ARM9TDMI 核的处理器），ARM940T（ARM9TDMI 核的处理器），StrongARM（Intel 公司的产品）。

5．V5 结构

V5 结构的 ARM 处理器提升了 ARM 和 Thumb 两种指令的交互工作能力，同时有了 DSP 指令-V5E 结构、Java 指令-V5J 结构的支持。

属于 V5T（支持 Thumb 指令）体系结构的处理器（核）有 ARM10TDMI，ARM1020T（ARM10TDMI 核处理器）。

属于 V5TE（支持 Thumb，DSP 指令）体系结构的处理器（核）有 ARM9E，ARM9E-S（ARM9E 可综合版本），ARM946（ARM9E 核的处理器），ARM966（ARM9E 核的处理器），ARM10E，ARM1020E（ARM10E 核处理器），ARM1022E（ARM10E 核的处理器），Xscale（Intel 公司产品）。

属于 V5TEJ（支持 Thumb，DSP 指令，Java 指令）体系结构的处理器（核）有 ARM9EJ，ARM9EJ-S（ARM9EJ 可综合版本），ARM926EJ（ARM9EJ 核的处理器），ARM10EJ。

6．V6 结构

V6 结构是在 2001 年发布的，在该版本中增加了媒体指令，属于 V6 体系结构的处理器核有 ARM11（2002 年发布）。V6 体系结构包含 ARM 体系结构中所有的 4 种特殊指令集：Thumb 指令（T）、DSP 指令（E）、Java 指令（J）和 Media 指令。

目前，基于 ARM 核结构的微处理器目前包括下面几个系列。

- ARM7 系列

ARM7 系列包括 ARM7TDMI、ARM720T、ARM7TDMI-S、ARM7EJ，该系列中，使用最广泛的是基于 ARM7TDMI 核的 ARM 处理器，比如 Samsung 的 S3c4510B、S3c44b0x 等，在这里后缀 TDMI 的含义如下。

T：表示支持 Thumb 指令集。

D：表示支持片上调试（Debug）。

M：表示内嵌硬件乘法器（Multiplier）。

I：表示支持片上断点和调试点。

- ARM9 系列

ARM9 系列包括 ARM920T、ARM922T 和 ARM940T。ARM9 处理器采用了 5 级流水线，指令执行效率较 ARM7 有较大提高，而且带有 MMU 功能，这也是与 ARM7 的重要区别。同时，该系列的处理器支持指令 Cache 和数据 Cache，因而具有更高的数据处理能力，主要应用在无线设备、手持终端、数字照相机等。

- ARM9E 系列

ARM9E 系列包括 ARM926EJ-S、ARM946E-S、ARM966E-S、ARM968E-S，该系列的处理器是综合类的处理器，它使用单一的处理器核提供了微控制器、DSP、Java 应用，因而非常适应于同时使用 DSP 和微控制器的场合。采用了 5 级流水线，支持 DSP 指令集、32 位的高速 AMBA 总线接口，带有 MMU 功能，最高主频可达 300MIPS。

- ARM10E 系列

ARM10E 系列包括 ARM1020E、ARM1022E、ARM1026EJ-S，该系列的 ARM 处理器采用了新的体系结构，同 ARM9 系列的相比有了很大的提高，采用了更高的 6 级流水线结构，支持 DSP 指令，适合同时需要高速数字信号处理的场合，支持 64 位的高速 AMBA 总线接口、32 位的 ARM 指令集和 16 位的 Thumb 指令集。主要应用于下一代的无线设备、数字消费品等。

- ARM11 系列

ARM11 系列包括 ARM1136J(F)-S，ARM1156T2(F)-S，ARM1176JZ(F)-S，AMR 公司在 2003 年推出了 ARM11 架构的核，基于 ARM11 核结构的处理器具有更高的性能，尤其是在多媒体处理能力方面，采用了先进的 0.13μm 工艺，最高工作频率可达 750MHz。

- SecurCore 系列

SecurCore 系列包括 SecurCore SC100、SecurCore SC110、SecurCore SC200 和 SecurCore SC22，SecurCor 系列处理器专为安全需要而设计，提供了对于安全方案解决的支持，主要应

用在比如电子商务、电子银行、网络认证等对安全性要求很高的场合。

- Inter 的 Xscale

Xscale 处理器是 Intel 公司基于 ARMV5TE 体系结构的解决方案，是一款高性能、低功耗的 32 位 RISC 处理器，有 PXA25x 系列和 PXA27x 系列，相比较早期的 StrongARM 处理器，Xscale 处理器是 Intel 公司目前主推的 ARM 处理器，主要应用在 PDA 和网络产品等方面。

2.1.3 Linux 与 ARM 处理器

在 32 位 RISC 处理器领域，基于 ARM 的结构体系在嵌入式系统中发挥了重要作用，ARM 处理器和嵌入式 Linux 的结合也正变得越来越紧密，并在嵌入式领域得到了广阔的应用。早在 1994 年，Linux 就可在 ARM 架构上运行，但那时 Linux 并没有在嵌入式系统中得到太多应用。目前，上述状况已经出现巨大变化，包括便携式消费类电子产品、网络、无线设备、汽车、医疗和存储产品在内，都可以看到 ARM 与 Linux 相结合的身影，linux 之所以能在嵌入式市场上取得如此辉煌的成就，与其自身的优秀特性是分不开的。

Linux 具有诸多内在优点，非常适合于嵌入式操作系统。

- Linux 的内核精简而高效，针对不同的实际需求，可将内核功能进行适当地剪裁，Linux 内核可以小到 100KB 以下，减少了对硬件资源的消耗。
- Linux 诞生之日就与网络密不可分，它本身就是一款优秀的网络操作系统，Linux 具有完善的网络性能，并且具有多种网络服务程序，而操作系统具备网络特性是很重要的。
- Linux 的可移植性强，方便移植到许多硬件平台，其模块化的特点也便于开发人员进行删减和修改，同时，Linux 还具有一系列优秀的开发工具，嵌入式 Linux 为开发者提供了一整套的工具链（Tool Chain），能够很方便地实现从操作系统内核到用户态应用软件各个级别的调试。
- Linux 源码开放，软件资源丰富，目前可以支持多种硬件平台，如 X86、ARM、MIPS 等，目前已经成功移植到数十种硬件平台之上，几乎包括所有流行的 CPU 架构，同时 Linux 下面有着非常完善的驱动资源，支持各种主流硬件设备，所有这些都促进了 Linux 在嵌入式领域广泛的应用。

不同特征的 Linux 都是在某一个 CPU 架构体系上运行的，而 ARM 结构体系历经多年的发展产生出很多版本，Linux 对于已在 ARM 规划蓝图中获定义的新特征也有相应的支持。ARM 体系的处理器按照不同的目标应用分类有着不同的特点和发展方向，基于与操作系统结合的特点考虑，可以根据有无 MMU（Memory Management Unit）把 CPU 分成两类，即带 MMU 功能的处理器和不带 MMU 功能的处理器。

Linux 作为一种基于 X86 平台发展过来的操作系统，是一种典型的应用操作系统，在硬件上需要 MMU 的支持，所以只有在包含 MMU 的 ARM 处理器上才能运行 Linux，如典型的 ARM720T、ARM920/922T 和 ARM926EJ。另外一些常用的 ARM 处理器，如 ARM7TDMI 系列，因为没有 MMU，所以不支持标准的 Linux。不带 MMU 的处理器由于特别适合于深度嵌入的特点（如快速实时响应、实地址编程等），在嵌入式系统中的应用非常广泛。为了适应

这种需求，uClinux 应运而生。uClinux 是开放源码的嵌入式 Linux 的一个经典之作，它设计的目标平台是那些没有内存管理单元（MMU）的微处理器芯片。为了满足嵌入式系统的需求，uClinux 还改写和裁减了大量 Linux 内核代码，因此 uClinux 内核远小于标准 Linux 的内核，但仍然保持了 Linux 操作系统的绝大部分特性，包括稳定强大的网络功能及出色的文件系统支持等。

目前，在 2003 年末推出的新版本的 Linux 内核 2.6 版本加强了对无 MMU 处理器的的支持，Linux 2.6 内核扩展多嵌入式平台支持的一个主要途径就是把 uCLinux 的大部分并入主流内核功能中，这无疑为 Linux 在嵌入式领域的广泛应用加重了砝码，也使得 ARM 与 Linux 的关系更加紧密。

2.2 ARM 指令集

2.2.1 ARM 微处理器的指令集概述

ARM 微处理器的指令集主要有 6 大类。

- 跳转指令
- 数据处理指令
- 程序状态寄存器（PSR）处理指令
- 加载/存储指令
- 协处理器指令
- 异常产生指令

具体的指令及功能如表 2.2 所示（表中指令为基本 ARM 指令，不包括派生的 ARM 指令）。

表 2.2　ARM 指令及其功能描述

指　　令	指令功能描述
ADC	带进位加法指令
ADD	加法指令
AND	逻辑与指令
B	跳转指令
BIC	位清零指令
BL	带返回的跳转指令
BLX	带返回和状态切换的跳转指令
BX	带状态切换的跳转指令
CDP	协处理器数据操作指令
CMN	比较反值指令

CMP	比较指令
EOR	异或指令
LDC	存储器到协处理器的数据传输指令
LDM	加载多个寄存器指令
LDR	存储器到寄存器的数据传输指令
MCR	从 ARM 寄存器到协处理器寄存器的数据传输指令
MLA	乘加运算指令
MOV	数据传送指令
MRC	从协处理器寄存器到 ARM 寄存器的数据传输指令
MRS	传送 CPSR 或 SPSR 的内容到通用寄存器指令
MSR	传送通用寄存器到 CPSR 或 SPSR 的指令
MUL	32 位乘法指令
MVF	传送值到浮点数寄存器
MVN	数据取反传送指令
ORR	逻辑或指令
RSB	逆向减法指令
RSC	带借位的逆向减法指令
SBC	带借位减法指令
STC	协处理器寄存器写入存储器指令
STM	批量内存字写入指令
STR	寄存器到存储器的数据传输指令

续表

指　　令	指令功能描述
SUB	减法指令
SWI	软件中断指令
SWP	交换指令
TEQ	相等测试指令
TST	位测试指令

2.2.2　ARM 指令寻址方式

1．立即数寻址

ARM 指令的立即数寻址是一种特殊的寻址方式，操作数本身就在指令中给出，只要取

出指令也就取到了操作数。这个操作数被称为立即数。

```
ADD R0，R0，＃1            ; R0←R0 + 1
ADD R0，R0，＃0x3A         ; R0←R0 + 0x3A
```

在以上 2 条指令中，第 2 个源操作数即为立即数，实际使用时以“＃”符号为前缀。

2．寄存器寻址

寄存器寻址就是利用寄存器中的数值作为操作数，这种寻址方式是各类微处理器经常采用的一种方式，也是一种执行效率较高的寻址方式。如以下的指令。

```
ADD R0，R1，R2             ; R0←R1 + R2
```

该指令的执行效果是将寄存器 R1 和 R2 的内容相加，其结果存放在寄存器 R0 中。

3．寄存器间接寻址

寄存器间接寻址就是以寄存器中的值作为操作数的地址，而操作数本身存放在存储器中。例如以下指令。

```
ADD R0，R1，[R2]           ; R0←R1 + [R2]
LDR R0，[R1]               ; R0←[R1]
```

在第 1 条指令中，以寄存器 R2 的内容作为操作数的地址，然后与 R1 相加，其结果存入寄存器 R0 中。

第 2 条指令将以 R1 的值为地址的存储器中的内容送到寄存器 R0 中。

4．基址变址寻址

基址变址的寻址方式就是将寄存器（该寄存器一般称作基址寄存器）的内容与指令中给出的地址偏移量相加，从而得到一个操作数的有效地址。如下面的几条指令所示。

```
LDR R0，[R1，＃0x0A]          ; R0←[R1 + 0x0A]
LDR R0，[R1，＃0x0A]!         ; R0←[R1 + 0x0A]、R1←R1 + 0x0A
```

在第 1 条指令中，将寄存器 R1 的内容加上 0x3A 形成操作数的有效地址，将该地址处的操作数送到寄存器 R0 中。

在第 2 条指令中，将寄存器 R1 的内容加上 0x0A 形成操作数的有效地址，从而取得操作数存入寄存器 R0 中，然后，R1 的内容自增 0x0A 个字节。

5．多寄存器寻址

采用多寄存器寻址方式，一条指令可以完成多个寄存器值的传送。这种寻址方式可以用一条指令完成传送最多 16 个通用寄存器的值。比如下面的指令。

```
LDMIA R0，{R1，R2，R3，R4}        ; R1←[R0]
                                  ; R2←[R0 + 4]
```

```
                                        ; R3←[R0 + 8]
                                        ; R4←[R0 + 12]
```

该指令的后缀 IA 表示在每次执行完加载/存储操作后，R0 按字长度增加，因此，指令可将连续存储单元的值传送到 R1～R4。

6．相对寻址

与基址变址寻址方式相类似，相对寻址以程序计数器 PC 的当前值为基地址，指令中的地址标号作为偏移量，将两者相加之后得到操作数的有效地址。比如下面的程序段完成子程序的调用和返回，跳转指令 BL 采用了相对寻址方式。

```
    BL   NEXT                           ; 跳转到子程序 NEXT 处执行
    ……
NEXT
    ……
    MOV  PC, LR                         ; 从子程序返回
```

7．堆栈寻址

堆栈是一种数据结构，按先进后出（First In Last Out，FILO）的方式工作，使用一个称作堆栈指针的专用寄存器指示当前的操作位置，堆栈指针总是指向栈顶。

根据堆栈的生成方式，堆栈又可以分为递增堆栈（Ascending Stack）和递减堆栈（Decending Stack），当堆栈由低地址向高地址生成时，称为递增堆栈；当堆栈由高地址向低地址生成时，称为递减堆栈。这样就有 4 种类型的堆栈工作方式，ARM 微处理器支持以下 4 种类型的堆栈工作方式。

（1）满递增堆栈：堆栈指针指向最后压入的数据，并且堆栈以递增方式向上生成。

（2）满递减堆栈：堆栈指针指向最后压入的数据，并且堆栈以递减方式向下生成。

（3）空递增堆栈：堆栈指针指向下一个将要放入数据的空位置，且由低地址向高地址生成。

（4）空递减堆栈：堆栈指针指向下一个将要放入数据的空位置，且由高地址向低地址生成。

2.2.3 Thumb 指令概述

作为 32 位的嵌入式处理器，ARM 具有 32 位数据总线宽度，但是为了更好地兼容数据总线宽度为 16 位的应用系统，ARM 体系结构除了支持执行效率很高的 32 位 ARM 指令集以外，同时支持 16 位的 Thumb 指令集。Thumb 指令集是 ARM 指令集的一个子集，允许指令编码为 16 位的长度。与等价的 32 位代码相比较，Thumb 指令集在保留 32 代码优势的同时，可以在很大程度上节省系统的存储空间。当处理器在执行 ARM 程序段时，称 ARM 处理器处于 ARM 工作状态；当处理器在执行 Thumb 程序段时，称 ARM 处理器处于 Thumb 工作状态。

所有的 Thumb 指令都有对应的 ARM 指令，而且 Thumb 的编程模型也对应于 ARM 的编程模型，在应用程序的编写过程中，只要遵循一定调用的规则，Thumb 子程序和 ARM 子程

序就可以互相调用，二者结合应用可以充分发挥各自的特点，取得较好的效果。

2.3 典型 ARM 处理器简介

2.3.1 Atmel AT91RM9200

Atmel 公司的 32 位 RISCC 处理器 AT91RM9200 是基于 ARM Thumb 的 ARM920T（核）微控制器，时钟频率为 180MHz，运算速度可以达到 200MIPS。带有全性能的 MMU，支持 SDRAM、静态存储器、Burst Flash、CompactFals、SmartMedia 以及 NAND Flash，具有高性能、低功耗、低成本、小体积等优点。AT91RM9200 微处理器是一个多用途的通用芯片，它内部集成了微处理器和常用外围组件，具有更高性价比的特点，可以为工控领域嵌入式系统提供优秀的解决方案。

AT91RM9200 具有以下的丰富片上资源。

（1）16KB 数据 Cache，16KB 指令 Cache；

（2）虚拟内存管理单元 MMU；

（3）带有 Debug 调试的在片 Emulator；

（4）Mid-level Implementation Embedded Trace Macrocell；

（5）16KB 的内部 SRAM 和 128KB 的内部 ROM；

（6）带有外部总线接口（EBI），方便用户进行扩展升级；

（7）支持 SDRAM、SRAM、Burst Flash 和 CompactFlash、SmartMedia and NAND Flash 的无缝连接；

（8）增强型的时钟产生器和电源管理单元；

（9）带有 2 个 PLL 的 2 个在片振荡器；

（10）慢速的时钟操作模式和软件电源优化能力；

（11）4 个可编程的外部时钟信号；

（12）包括周期性中断、看门狗和第二计数器的系统定时器；

（13）带有报警中断的实时时钟；

（14）带有 8 个优先级、可单个屏蔽中断源、Spurious 中断保护的先进中断控制器；

（15）7 个外部中断源和 1 个快速中断源；

（16）4 个 32 位的 PIO 控制器，可以达到 122 个可编程 I/O 引脚（每个都有输入控制、可中断及开路的输出能力）；

（17）20 通道的外部数据控制器（DMA）；

（18）10/100M 的以太网接口；

（19）2 个全速的 USB 2.0 主接口和一个从口；

（20）4 个 UART；

（21）3 通道 16 位的定时/计数器（TC）；

（22）两线接口（TWI）；

（23）IEEE 1149.1 JTAG 标准扫描接口。

2.3.2 Samsung S3C2410

S3C2410 是著名的半导体公司 Samsung 推出的一款 32 位 RISC 处理器，为手持设备和一般类型的应用提供了低价格、低功耗、高性能微控制器的解决方案。S3C2410 的内核基于 ARM920T，带有 MMU（Memory Management Unit）功能，采用 0.18μm 工艺，其主频可达 203MHz，适合于对成本和功耗敏感的需求，同时它还采用了 AMBA（Advanced Microcontroller Bus Architecture）的新型总线结构，实现了 MMU、AMBA BUS、Harvard 的高速缓冲体系结构，同时支持 Thumb16 位压缩指令集，从而能以较小的存储空间需求，获得 32 位的系统性能。

其片上功能如下。

（1）内核工作电压为 1.8/2.0V、存储器供电电压 3.3V、外部 I/O 设备的供电电压 3.3V；

（2）16KB 的指令 Cache 和 16KB 的数据 Cache；

（3）LCD 控制器，最大可支持 4K 色 STN 和 256 色 TFT；

（4）4 通道的 DMA 请求；

（5）3 通道的 UART（IrDA1.0、16 字节 TxFIFO、16 字节 RxFIFO），2 通道的 SPI 接口；

（6）2 通道的 USB（Host/Slave）；

（7）4 路 PWM 和 1 个内部时钟控制器；

（8）117 个通用 I/O，24 路外部中断；

（9）272Pin FBGA 封装；

（10）16 位的看门狗定时器；

（11）1 通道的 IIC/IIS 控制器；

（12）带有 PLL 片上时钟发生器。

S3C2410 ARM 处理器支持大/小端模式存储字数据，其寻址空间可达 1GB，每个 Bank 为 128MB，对于外部 I/O 设备的数据宽度，可以是 8/16/32 位，所有的存储器 Bank（共有 8 个）都具有可编程的操作周期，而且支持各种 ROM 引导方式（NOR/NAND Flash、EEPROM 等），其结构框图如图 2.1 所示。

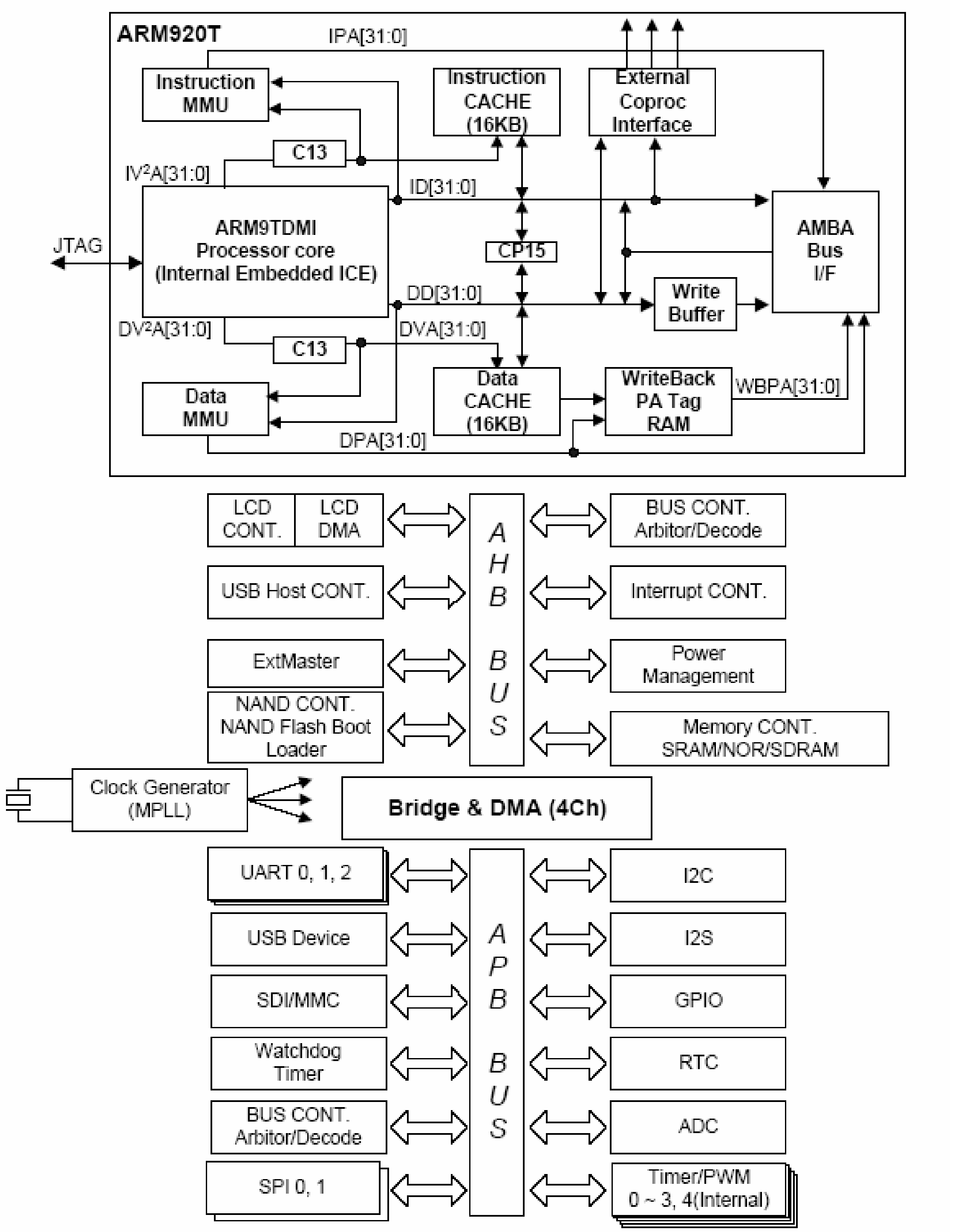


图 2.1　S3C2410 结构框图

2.3.3 TI OMAP1510/1610 系列

TI 在 1998 年推出了可扩展的开放式 OMAP 处理器平台，OMAP 平台提供了语音、数据和多媒体所需的带宽和功能，可以极低的功耗为高端 2.5G 和 3G 无线设备提供较高的性能。TI 的 OMAP 处理器支持所有类似的高级操作系统，无需任何新的编程技能便可提供无缝访问其高性能 DSP 算法的能力。TI 还提供了 OMAP 解决方案，将无线调制解调器与专用应用处理器完美地组合在单个芯片上。TI 在提供全球范围的技术支持的同时，还提供了可降低系统成本的高度集成的解决方案。

OMAP（开放式多媒体应用平台）是 TI 公司针对无线市场推出的一系列针对便携设备的多媒体处理器，但其应用并不限于手机。OMAP 系列处理器一般具有双核（DSP 和 ARM）架构，这种低功耗的 OMAP 架构把用于语音的 DSP 信号处理功能与 RISC 处理器的通用系统性能融合在了一起，设计了开放式软件架构，以鼓励开发语音引擎、语音应用和多媒体等应用，包括语音识别器和原型应用等开发支持，可帮助开发商快速建立其自己的产品并缩短产品上市时间。除具有“性能/功耗比”上的优势之外，OMAP 系列处理器还提供丰富的外围接口，支持几乎所有流行的有线和无线接口标准。

按功能应用来区分，TI 的 OMAP 处理器主要分为两类，如表 2.3 所示。

表 2.3　　OMAP 系列分类

类　别	型　号	描　述
单纯应用处理器	OMAP310	175MHz，具有基本的多媒体功能
	OMAP1510	175MHz，与 OMAP310 相比增加了 DSP 和 ARM926 处理内核，192KB 的片内 RAM
	OMAP1610	204MHz，功耗和封装尺寸较 1510 更小，多媒体处理能力有较大提高，增加了 JAVA 加速器，采用硬件方法加速应用程序的执行，集成了更多运动控制和接口器件
	OMAP1611	204MHz，增加了内部 SRAM（有助于提高流媒体和图形处理能力）和 54Mbit/s 的 WLAN 接口
	OMAP1612	204MHz，较 1611 又增加了堆叠式整合的 DDR 存储器，与外接存储器方式相比减少了空间和功耗
整合数字基带功能的应用处理器	OMAP710	132MHz，针对中端智能手机，性能与 1510 对应
	OMAP730	200MHz，性能较 710 提高一倍，待机时间也增加一倍，是 TI 目前主推的芯片
	OMAP732	200MHz，与 730 相似，但将 SRAM 以堆叠式整合，减小了体积

OMAP1510 是一颗双核心的基于 2.5G/3G 手持设备和 PDA 产品应用的多媒体应用芯片，这种芯片包括了完整的视频流媒体处理功能、音频解码功能、移动通信协议的，面向 OEM/

ODM 客户的产品。该芯片由 2 部分构成：TMS320C55x DSP 芯片和扩展型的 ARM925 芯片。其中 ARM925 芯片负责控制部分的功能、操作系统的用户界面接口支持。而 TMS320C55x 则负责安全性、多媒体和语音方面的处理。这种独特的双核心架构把高性能低功耗的 DSP 核与控制功能很强的 ARM 处理器结合起来，具有集成度高、硬件可靠性和稳定性好、速度快、数据处理能力强、功耗低、开放性好等优点。

为了适应 3G 的应用发展，TI 又推出了新的应用处理器 OMAP1610/1611/1612 系列。新的 OMAP 处理器对安全应用、Java、多媒体和图形处理均采用了硬件加速器，并且还预留了 802.11a/b/g 接口。OMAP161X 系列的处理器主要具有以下特点。

1．低功耗、高性能 CMOS 技术

- 采用低电压工作模式，内核 1.1～1.5V，I/O1.8～3V。
- 静态消耗电流小于 120μA。
- 优化了时钟和电源管理，只需要 13MHz 和 32kHz 的两个时钟。
- 0.13μm 工艺技术。
- 12mm×12mm BGA 封装。

2．TMS320C55xDSP 核

- 最高工作频率可达 204MHz。
- 带有片内 32K×16 位的双口 RAM（DARAM）（64KB）。
- 48K×16 位的片内 RAM（SARAM）（96KB）。
- 24KB 的指令 cache。
- 每时钟周期执行单/双指令。

3．ARM926TEJ 核

- 最高 204MHz 的工作频率，采用 ARM926TEJ V5 版本架构。
- 16KB 的指令 Cache，8KB 的数据 Cache。
- 采用了 Java 加速器。
- 支持 32 位和 16 位（Thumb）指令结构。
- MMU 功能。
- 最高 204MHz 的工作频率，采用 ARM926TEJ V5 版本架构。
- 16KB 的指令 Cache，8KB 的数据 Cache。
- 采用了 Java 加速器。
- 支持 32 位和 16 位（Thumb）指令结构。

图 2.2 是 OMAP1610 处理器的结构框图。

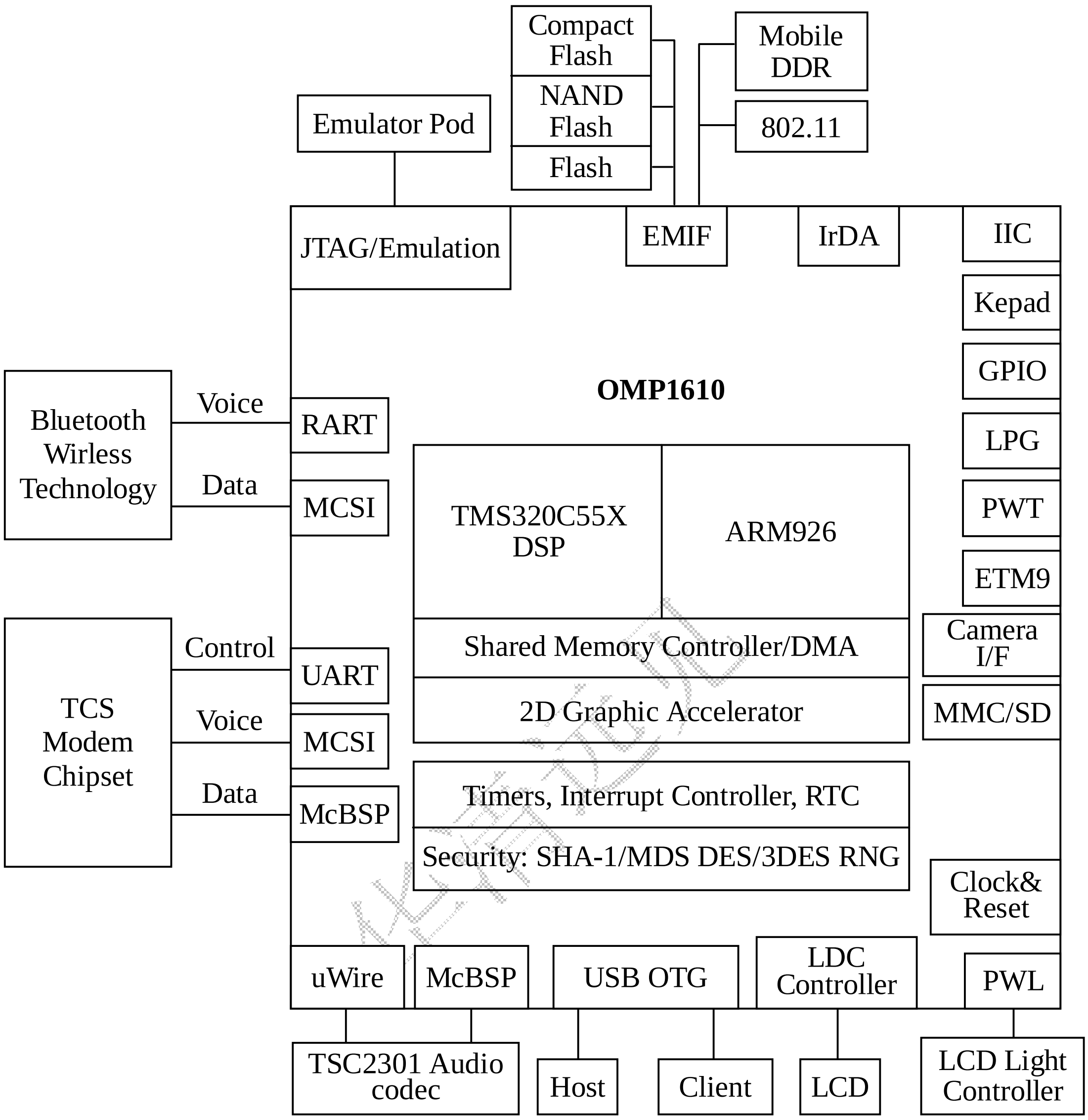


图 2.2　OMAP1610 结构框图

2.3.4　Freescale i.Max21

作为无线应用半导体领域的领导者，飞思卡尔半导体（其前身属于摩托罗拉半导体公司）开发了 i.MX 系列处理器，其 i.MX 系列嵌入式应用处理器采用了 ARM 内核，主流应用处理器 i.MAX21（第二代的龙珠芯片）在多媒体和安全性能、并行处理能力等方面都有不错的表现。i.MX21 可支持实时 MPEG4 和 H.263 编解码，最高可每秒传输 30 帧 CIF 或 QVGA 图像。另外，i.MX21 还支持 Mobile Java 3D 和 OpenGL-ES 等先进的图型软件标准，以及 Superscape、HI Corp 和 Fathammer 3D 软件引擎。可用于智能电话、无线个人数字助理（PDA）和许多其他移动产品。i.MX 应用处理器的组件数量少，电池寿命长，并且性能出众，方便开发出功能更加齐全（例如数字图像捕捉、文件共享、无线连接和多媒体娱乐）、经济更高效的支持无线

的手持设备。开发商采用这些平台可以开发出从以语音为主的 2G 手机到具有丰富功能和特色的 3G 手机，进而保持了开发商产品开发的延续性，并能很容易地实现开发平台的转换。

i.Max21 主要具有如下特性。

- 集成 ARM926 内核。
- 16KB 的指令 Cache 和 16KB 的数据 Cache。
- 采用 Smart Speed Switch 技术，可实现数据的并行处理，增加数据的吞吐量。
- 16/18 位的彩色 LCD 控制器，支持 SVGA。
- USB On-the-Go，2 通道的 USB Host。
- 支持实时 MPEG4 和 H.263 编解码，最高可每秒传输 30 帧 CIF 或 QVGA 图像。

其结构框图如图 2.3 所示。

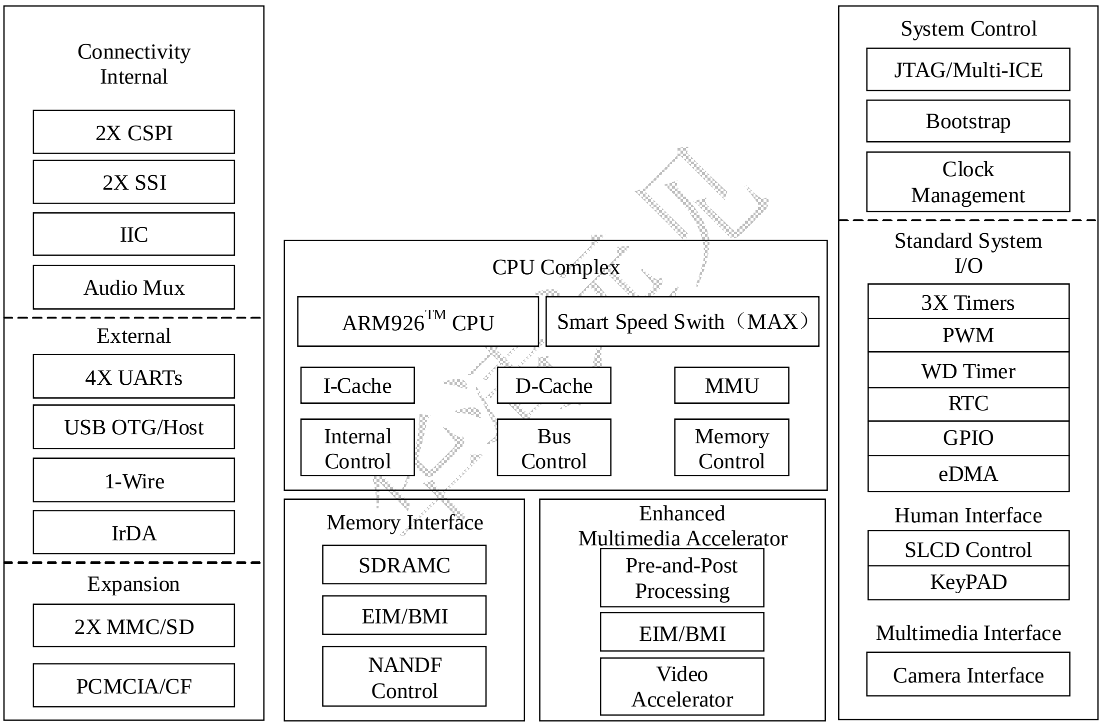


图 2.3　i.Max21 结构框图

2.3.5　Intel Xscale PXA 系列

Intel 的 PXA 处理器最早被称为 XScale 处理器。XScale 也是 ARM 处理器的一种衍生，不过它在架构扩展的基础上保留了对以往软件的向下兼容性。

Intel 目前开发的基于 ARM 核的处理器有 2 个系列。

- StrongARM-StrongARM SA1100；
- 基于 Xscale 架构的 ARM 处理器 Xscale PXA 系列。

Xscale 是一款功耗低、伸缩度高的产品，并且其最大的优势就是核心频率可以高速的提升。此外，Xscale 整合了以往其他 ARM 处理器所不会去整合的多媒体指令集——Wireless MMX，这种指令集类似桌面处理器的多媒体指令集，是一种 64bit 的精简指令，这种指令集可以大大地优化视频播放、3D 图像显示、音频处理等应用，同时这种指令集也会大大降低程序开发者的开发难度，从而加快开发进度。

基于 Xscale 架构的 ARM 处理器，Intel 到目前为止一共开发 3 个系列的处理器。

- PXA25x 系列；
- PXA26x 系列；
- PXA27x 系列。

这其中 PXA25x 是最早一代的产品，一经推出就获得了很大的成功，其工作频率可以为 200、300、400MHz，使用了较新的 0.18μm 工艺制程、内含 32KB 的指令缓存，32KB 数据缓存以及多媒体流数据专用 2KB 缓存；最高支持 256MB 的内存、包含双通道 PCMCIA、CF 卡控制器、MMC/SD 控制器；包含 LCD 显示控制器、AC97 音频、USB 接口、红外接口、蓝牙接口；芯片采用 256Pin 的 PBGA 封装。PXA27x 系列处理器主要有 3 个成员：PXA270、PXA271、PXA272。每个成员中都有 312MHz、416MHz、520MHz、624MHz 这几种 CPU 主频的产品。其中，PXA271 内部集成了 32MB 的 Nor Flash（16bit 数据线）和 32MB 的 SDRAM（16bit 数据线）资源；PXA272 内部集成了 64MB 的 Nor Flash（32bit 数据线），这是很多 ARM 处理器所不具备的。

PXA270 是 Intel 继 PXA250/PXA255/PXA260 之后，于 2004 年 4 月发布的最新款 XScale 处理器家族的升级产品，最高主频达 624MHz。该款芯片把 X86 架构奔腾 4 系列上的多媒体扩展功能引入了 Xscale 芯片组的产品线中，用户通过这个无线多媒体扩展技术（MMX）可以在掌上设备上播放高质量的视频和运行三维游戏。同时 PXA270 还加入了 Intel SpeedStep 动态电源管理技术，在保证 CPU 性能的情况下，能够最大限度地降低移动设备的功耗，广泛应用在高端 PDA，智能手机，网络路由器、无线通信和控制系统等嵌入式系统开发。

PXA270 内置了 Intel 的无线 MMX 技术，显著提高了多媒体性能，312MHz 的 CPU（PXA270 系列中最低钟频的产品）将达到 520MHz ARM CPU 的多媒体处理效能，而主频达到 624MHz，Intel 公司同时还发表了配合 PXA270 使用的图形协处理器——2700G 多媒体加速器；这颗芯片可以以每秒 30 帧的速度播放 MPEG4 或 WMV 的图像，使 PXA270 的多媒体性能达到极大提升。

图 2.4 所示是 PXA270 处理器的内部结构框图。

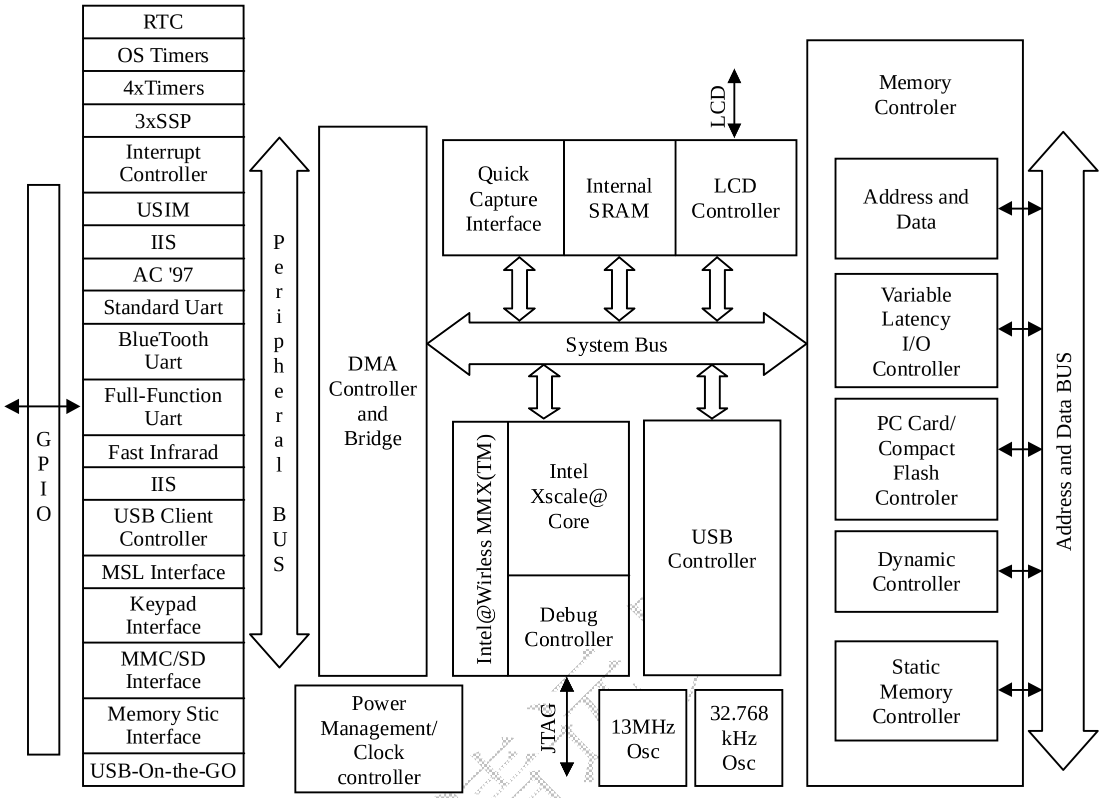


图 2.4　PXA270 内部结构框图

2.4　三星 S3C2410 开发板

2.4.1　三星 S3C2410 开发板介绍

在读者准备研究嵌入式开发时，选择合适的开发平台也是一个很重要的环节。开发板可以为用户提供基本的底层硬件、系统和驱动等资源。目前，针对同一处理器的开发板有很多种，当然，开发目的是各不相同的，其中要考虑的因素也很多，诸如开发成本、开发板资源是否满足要求、周期、技术支持程度，等等。本书以深圳优龙科技有限公司的 ARM9 开发套件 FS2410 开发版作为研究平台来进行介绍，该开发板是基于 Samsung 2410 处理器为核心的嵌入式开发平台，提供了完备的软硬件资源，为广大嵌入式开发爱好者深入研究嵌入式开发技术提供了一个较好的平台。

FS2410 开发板的外观如图 2.5 所示。

该开发板的资源如下。

1．硬件资源情况

- CPU：三星 S3C2410A，主频 203MHz。

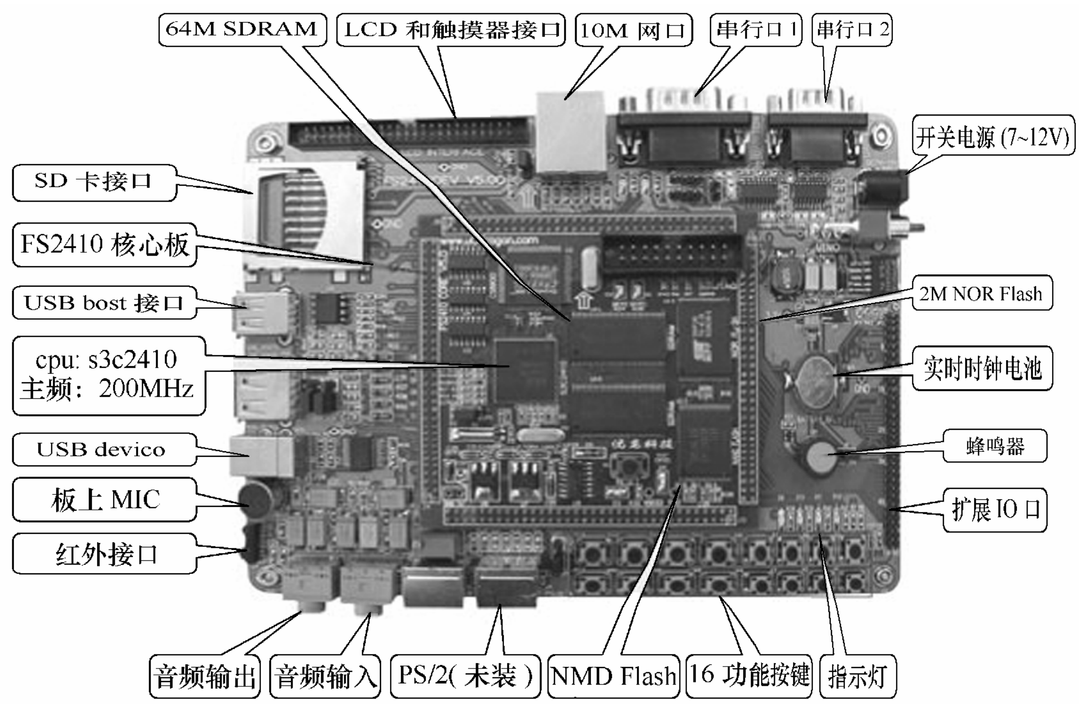


图 2.5　FS2410 开发板

- 内存：64MB。
- NOR Flash：2MB（SST39VF160 或 SST39VF1601）。
- NAND Flash：64MB（K9F1208，用户可自己更换为 16MB、32MB 或 128MB 的 NandFlash）。
- 2 个标准 5 线串口。
- 10M 网口，CS8900Q3，带联接和传输指示灯。
- 2 个 USB1.1 HOST 接口（其中一个 HOST 与 Device 复用，通过短路块选择）。
- 1 个 USB1.1 Device 接口（它与 USB HOST 接口复用，通过短路块选择）。
- 1 个 IRDA 红外线数据通信口。
- 采用 IIS 接口芯片 UDA1341，一路立体声音频输出接口可接耳机或音箱。
- 支持录音，自带驻机体话筒可直接录音，另有一路话筒输入接口可接麦克风。
- 1 个 SD 卡接口，可接 256M SD 卡。
- 1 个 50 芯 LCD 接口引出了 LCD 控制器和触摸屏的全部信号。
- 支持黑白、4 级灰度、16 级灰度、256 色、4096 色 STN 液晶屏，尺寸从 3.5 寸到 12.1 寸，屏幕分辨率可达到 800×600 像素。
- 支持黑白、4 级灰度、16 级灰度、256 色、64K 色、真彩色 TFT 液晶屏，尺寸从 3.5 寸到 12.1 寸，屏幕分辨率可达到 800×600 像素。

- 标准配置为夏普 256K 色 240×320/3.5 英寸 TFT 液晶屏，带触摸屏。
- 内部实时时钟（带有后备锂电池）。
- 1 个 20 芯 Multi-ICE 标准 JTAG 接口，支持 SDT2.51，ADS1.2 等调试。
- 开关电源供电，输入直流电压范围是 7～20V，带电源开关和指示灯。
- 1 个 EEPROM（AT24C02）用来验证 IIC 总线读写。
- 16 个小按键，4 个高亮 LED。
- 1 个蜂鸣器（带使能控制的短路块）。
- 2 个 PS/2 接口，信号线接在中断引脚上。
- 1 个精密可调电阻接到 ADC 引脚上用来验证模数转换。
- 1 个 60 芯 2mm 间距双排标准连接器用作扩展口，引出了地址线、数据线、读写、片选、中断、I/O 口、ADC、5V 和 3.3V 电源、地等用户扩展可能用到的信号。

2．软件资源情况

- ADS1.20 安装程序（评估版）。
- 采用 Linux2.4 以上内核。
    - 支持多种文件系统，比如 Cramfs、Fat 以及用于 NAND Flash 的 YAFFS 文件系统。
    - 支持 LCD 和触摸屏。
    - 支持 USB HOST。
    - 支持 QT。
    - 支持 MP3 播放和视频播放。
    - 支持多种网络应用，比如 FTP、HTTP、Telnet 等网络应用。
- 烧写 Flash 的工具软件 SJF2410（包含 NT/2000/XP 解决方案）。
- 串口工具软件 sscom32.exe、dnw.exe、tftp.exe。
- 64K 色（RGB565）图片字模软件。
- USB Device 接口驱动程序。
- FS2410 BIOS 源代码（ADS1.20 的项目文件）。
- FS2410 测试程序（ADS1.20 的项目文件，包含全部源代码）。
- Linux for S3c2410 内核源码包以及编译工具。
- WINCE4.2.NET 板级支持包 BSP for S3c2410。
- 已经编译好并可在 FS2410 上运行的 wince 内核，基于优龙提供的 BSP。
- Samsung 半导体网站关于 S3C2410 的全部资料和参考代码。
- FS2410 核心板和底板电路原理图。

2.4.2　众多的开发板供应商

随着嵌入式开发领域的不断深入和发展，现在越来越多的半导体厂商都开发了基于 ARM 核的 32 位 RISC 嵌入式处理器，并且为了推广各自的芯片，都有配套完善的开发板提供给用户。目前可以提供 ARM 芯片的著名欧美半导体公司有：Intel、德洲仪器、摩托罗拉、飞利浦半导体、意法半导体、ADI 公司、Atmel、Altera、CirrusLogic 等。日本的许多著名半导体公司如东芝、夏普、三菱半导体、爱普生、富士通半导体、松下半导体等公司较早期都大力

投入开发了自主的 32 位 CPU 架构的处理器，现在也已经转向购买 ARM 公司的 IP 进行新产品设计。韩国的现代半导体公司和著名的三星半导体也生产提供 ARM 芯片和相应的开发系统。我国台湾地区可以提供 ARM 芯片的公司有台积电、台联电、华邦电子等，大陆的公司如华为通讯和中兴通讯等公司也已经购买了 ARM 公司的相应知识产权，开始基于 ARM 核的芯片设计。

目前，国内的 ARM 开发板供应商大多与国外半导体厂商取得合作关系推出一系列的 ARM 开发板，其种类繁多，从面向低端的基本应用到高端设计的参考模型，应用尽有，除了前面提到的深圳优龙科技，还有华恒、英蓓特、傅立叶等，他们为用户提供了充足的选择余地以进行嵌入式的深入研究和二次开发。

"黑色经典"系列之《嵌入式 Linux 系统开发技术详解——基于 ARM》

FARSIGHT

第 3 章　Linux 编程环境

本章目标

本章内容包括常用的 Linux 开发工具使用技巧和 Linux 编程技术。本章内容比 Linux 编程方面的书籍简略得多，重点介绍常用的 Linux 编程工具和技巧。通过本章学习可以使读者快速掌握基本的 Linux 开发工具，为后续的嵌入式 Linux 开发打下基础。

- 常用 Linux 编程工具
- GNU 工具链的使用技巧
- Linux 编程库的 API 介绍

3.1 Linux 常用工具

3.1.1 Shell 简介

在 Linux 系统开发过程中，开发者或者用户与 Linux 系统（内核）进行交互的时候需要一个平台，这就是 Shell，有了它，用户就能通过键盘输入与系统进行交互了。Shell 会执行用户输入的命令，并且在屏幕上显示执行结果。这种交互的全过程都是基于文本方式的，这种面向命令行的用户界面被称为 CLI（Command Line Interface），在图形化用户界面（GUI）出现之前，人们一直是通过命令行界面来操作计算机的。Linux 的图形化环境最近这几年有很大改进，在 X 窗口系统下，只需打开 Shell 提示来完成极少量的任务。然而，许多 Linux 功能在 Shell 提示下要比在图形化用户界面（GUI）下完成得更加高效，况且一些应用程序并不支持图形界面。

单从字面意思上理解，Shell 的本意是“壳”的意思，通俗地讲就是内部核心与外部使用者发生联系的介质。当用户希望与系统内核（Kernel）发生联系进而控制硬件设备时，用户不会也不允许直接与内核交互，而必须通过 Shell 来下达命令使系统来控制硬件，同时内核也会通过 Shell 来反馈执行情况，这里的 Shell 就是一个桥梁。图 3.1 形象地说明了这一过程。

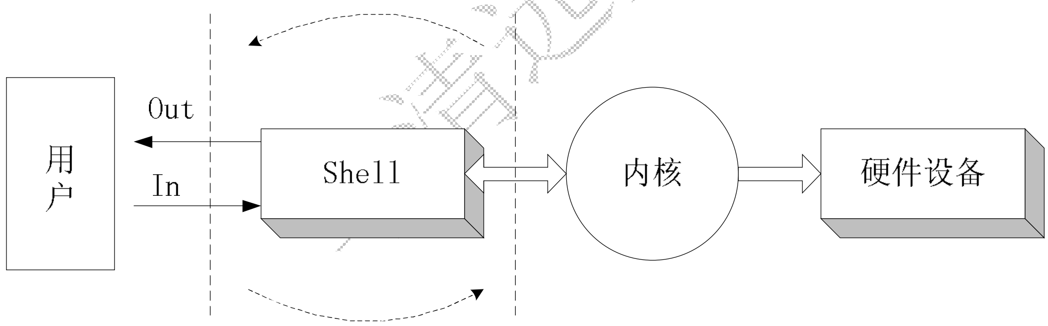


图 3.1 Shell 工作示意图

Shell 提供了用户与操作系统之间通讯的方式。这种通信可以以交互方式（从键盘输入，并且可以立即得到响应），或者以 Shell script（非交互）方式执行。Shell script 是放在文件中的一串 Shell 和操作系统命令，它们可以被重复使用。本质上，Shell script 是把命令行的命令简单地组合到一个文件中。

Shell 本身又是一个解释型的程序，也是一种编程语言，Shell 程序设计语言支持绝大多数在高级语言中能见到的程序元素，如函数、变量、数组和程序控制结构。Shell 编程语言简单而且易于掌握，任何在提示符中能键入的命令都能放到一个可执行的 Shell 程序中。作为操作系统的外壳，如果把 Linux 内核想像成一个系统的中心部分，那么 Shell 就是围绕内核的外层。当从 Shell 或其他程序向 Linux 传递命令时，内核会做出相应的反应。

历史上第一个真正的 Unix shell 称为“sh”，是 Stephen R. Bourne 于 20 世纪 70 年代中期

在新泽西的 AT&T 贝尔实验室编写的，为了纪念他，亦称为“Bourne shell”，Bourne Shell 是一个交换式的命令解释器和命令编程语言。在 20 世纪 80 年代早期，在美国 Berkeley 的加利福尼亚大学开发了 C shell（csh 和 tcsh），它主要是为了让用户更容易地使用交互式功能。C shell 是一种比 Bourne Shell 更适于编程的 shell，它的语法与 C 语言很相似。

Bash（Bourne Again Shell）是目前大多数 Linux（Red Hat，Slackware 等）系统默认使用的 Shell，它由 Brian Fox 和 Chet Ramey 共同完成，内部命令一共有 40 个，它是 Bourne Shell 的扩展，与 Bourne Shell 完全向后兼容，并且在 Bourne Shell 的基础上增加了很多特性。Bash 是 GNU 计划的一部分，用来替代 Bourne Shell。Linux 使用它作为默认的 Shell 是因为它有以下的特点。

- 可以使用类似 DOS 下面的 doskey 的功能，用上下方向键查阅和快速输入并修改命令。
- 自动通过查找匹配的方式，给出以某字串开头的命令。
- 包含了自身的帮助功能，只要在提示符下面键入 help 就可以得到相关的帮助。

Linux 下使用 Shell 非常简单，打开终端就可以看到 Shell 的提示符了，登录系统之后，系统将执行一个称为 Shell 的程序，正是 Shell 进程提供了命令行提示符。作为 Linux 默认的 Bash，对于普通用户用“$”作为 Shell 提示符，而对于根用户（root）用“#”作提示符。如图 3.2 所示。

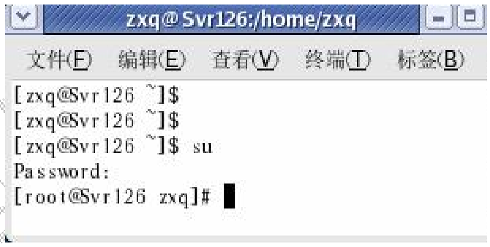


图 3.2 shell 提示符

从上面的界面中可以看到，当前用户是普通用户“zxq”时，Shell 提示符是‘$’；而当切换为根用户 root 时，Shell 提示符是‘#’。一旦出现了 Shell 提示符，就可以键入命令名称及命令所需要的参数了。用户键入有关命令行后，如果 Shell 找不到以其中的命令名为名字的程序，就会给出错误信息。例如，如果用户键入：

```
$mypfile
bash:myfile:command not found
$
```

可以看到，用户得到了一个没有找到该命令的错误信息。

3.1.2 常用 Shell 命令

目前，Linux 下基于图形界面的工具越来越多，许多工作都不必使用 Shell 就可以完成了。然而，专业的 Linux 使用者还是认为 Shell 是一个非常必要的工具，使用 Linux 时一定要熟悉 Sheel 的使用，至少要掌握一些基础知识和基本的命令。

由于 Bash 是 Linux 上缺省的 Shell，本章主要介绍 Bash 及其相关知识，Shell 命令可以分为两种。

- 包含于 Shell 内部的命令，如 cd 命令；
- 存在于系统文件内部的某个应用程序，如 ls 命令。

对用户使用 Shell 来说，不必关心一个命令是建立在 Shell 内部还是一个单独的程序。在

实际执行的时候，Shell 会首先检查输入的命令是否是 Shell 的内部命令，如果不是，再检查是否是一个内部的应用程序。然后 Shell 在搜索路径里寻找这些应用程序（搜索路径就是一个能找到可执行程序的目录列表）。如果键入的命令不是一个内部命令并且在路径里没有找到这个可执行文件，将会显示一条错误信息。如果能够成功找到命令，该内部命令或应用程序将被分解为系统调用并传给 Linux 内核。

Shell 命令的一般格式如下。

命令名 【选项】 【参数 1】 【参数 2】…

用户登录时，实际就进入了 Shell，它遵循一定的语法将输入的命令加以解释并传给系统。命令行中输入的第一个部分必须是一个命令的名字，第二个部分是命令的选项或参数，命令行中的每个部分必须由空格或 Tab 键隔开，注意，这里的选项和参数都用【】标注，这是说明它们都是可选的，因为有的命令不需要选项和参数就可以执行。

1．对于选项和参数的说明

【选项】是包括一个或多个字母的代码，它前面有一个减号（-），Linux 用它来区别选项和参数，【选项】可用于改变命令执行的动作的类型。多个【选项】可以用一个减号（-）连起来，例如‘ls -l-a’ 与‘ls -la’相同。

以常用的 ls 命令为例，ls 命令可以查看当前目录的内容，加入选项-l 可以以长格式查看当前目录内容，如图 3.3 所示.。

加入-l 选项，将会为每个文件列出一行信息，诸如数据大小和数据最后被修改的时间。

使用该指令可以查看文件的权限位，如上图中的“-rw-r--r--”符号，它表示的是 3 组不同用户对该文件的使用权限，每组有 3 个权限位，如下所示。

- rw- 用户权限



图 3.3 ls 命令

- r-- 同组用户权限
- r-- 其他用户权限

【参数】提供命令运行的信息，或者是命令执行过程中所使用的文件名。使用分号（;）可以将 2 个命令隔开，这样可以实现一行中输入多个命令。命令的执行顺序和输入的顺序相同。当然，ls 命令也可以加入参数，例如 ls -l /home/zxq 命令会将/home/zxq 目录的内容详细地列出。

2．命令行输入

命令行输入实际上是可以编辑的一个文本缓冲区，在命令行中就可以输入 Shell 命令了。在按“回车键”以确认当前操作之前，可以对输入的内容进行编辑。比如删除、复制、粘贴等，还可以插入字符，使得用户在输入命令，尤其是复杂命令时，若出现键入错误，无须重新输入整个命令，只要利用编辑操作，即可改正错误。

Bash 可以保存以前键入命令的列表，这一列表被称为命令历史表。按向上箭头键，便可以在命令行上逐次显示各条命令。同样，按向下箭头键可以在命令列表中向下移动，这样可以将以前的各条命令显示在命令行上，用户可以修改并执行这些命令，这样可以不用重复输入以前执行的命令。

3．常用 Shell 命令介绍

Shell 命令种类很多，功能也很复杂，下面主要就几种常用的 Shell 命令来介绍。

（1）输入命令行自动补齐（automatic command line completion）功能

在 Linux 下有时比如对文件操作的时候，有的文件名或文件夹的名称可能会很长，完全逐字输入比较麻烦，在输入命令的任何时刻，可以按<Tab>键，当这样做时，系统将试图补齐此时已输入的命令。例如，假设当前目录下有一文件：Busybox-pre-1.00.tar.gz，现在想要解压该文件，而该文件是当前目录下惟一以 B 开头的文件名，此时就可以如下操作。

```
# tar zxvf  B<Tab>usy-pre-1.00.tar.gz
```

此时，系统会自动补齐该文件名后面的部分，这样用起来就会非常方便。

使用命令行自动补齐功能，对于使用长命令或操作较长名字的文件或文件夹都是非常有意义的。

（2）对目录和文件的操作

- 改变当前目录

```
# cd  [目的目录名]
```

这里的目的目录名可用相对路径表示，也可以用绝对路径表示。如果要切换到上一级目录，可以采用下面的命令。

```
# cd ..
```

- 显示当前所在目录

Linux 下 pwd 命令是最常用的命令之一，用于显示用户当前所在的目录。例如：

```
# pwd
/home/TH
```

执行 pwd 指令后，系统提示当前所在的目录是/home/TH。

- 创建目录

在 Linux 下可以使用 mkdir 指令来创建一个目录。

```
# mkdir [新目录名]
```

例如：mkdir /home/TH，改命令的功能就是在/home/目录下创建 TH 子目录。

- 删除一个目录/文件

rm [选项] 被删除的文件/目录

对于选项的说明如下。

-r：完全删除目录，就其下的目录和文件也一并删除。

-i：在删除目录之前需要经过使用者的确认才能被删除。

-f：不需要确认就可以删除，也不会产生任何错误信息。

例如：rm –rf /home/TH/tmp，就是不必经过确认就把/home/TH/tmp/下的目录和文件全部删除。

- 拷贝文件/目录

```
# cp [ 选项] [源文件/目录] [指定文件/目录]
```

对于选项的说明如下。

-i:采用-i 选项时，当指定目录下已存在被复制的文件时，会在复制之前要求确认是否要覆盖，如使用者的回答是 y（yes）才执行复制的动作。

-p：保留权限模式和更改时间。

-r：此参数是用来将一目录下的所有文件都复制到另一个指定目录中。

例如：cp /etc/ld.conf ～/，拷贝/etc/目录下的 ld.conf 文件到系统的主目录中；

cp -r dir1 dir2，将目录 dir1 的全部内容全部复制到目录 dir2 里面。

- 建立文件的符号链接

建立文件的符号链接是 Linux 中一个很重要的命令，它的基本功能是为某一个文件在另外一个位置建立一个不同的链接，这个命令最常用的选项是-s，具体用法如下。

```
# ln [-s] [源文件] [目标文件]
```

在实际的操作过程当中，有时在不同的目录中要用到相同的文件，我们不需要在每一个需要的目录下都放一个相同的文件，而是使用 ln 命令链接（link）它就可以（相当于建立了一个快捷方式），这样可以避免重复的占用磁盘空间。例如：ln –s /bin/test /usr/local/bin/test，这就为/bin 下的 test 文件在/usr/local/bin 目录下建立了一个符号链接。

注意 使用 ln 命令需要注意: ln 命令会保持每一处链接文件的同步性，也即是说如果改动了某一文件，其他的符号链接文件都会发生相应的变化; 其次，ln 命令的链接方式又有软链接和硬链接两种，上面提到的用法就是软链接，它只会在你选定的位置上生成一个文件的镜像，不会占用磁盘空间，硬链接没有选项-s，它会在指定的位置上生成一个和源文件大小相同的文件，无论是软链接还是硬链接，文件都保持同步变化。

- 改变文件/目录访问权限

在 Linux 系统下面，一个文件有可读（r）、可写（w）、可执行（x）3 种模式，chmod 可以用数字来表示该文件的使用权限，其语法如下。

```
# chmod  [XYZ]  文件
```

其中 X、Y、Z 各为一个数字，分别表示 User（用户）、Group（同组用户）及 Other（其他用户）对于该文件的使用权限。对于文件的属性，r（可读）=4，w（可写）=2，x（可执行）=1。对于每一位用户来说，若要具有 rwx 属性则对应的位应为 4+2+1=7，若要 rw-属性则为 4+2=6，若要 r-x 属性则为 4+1=5。比如下面的例子：

```
# chmod  751  /home/TH/test
```

其执行结果就是使程序 test 对于用户可读、写、执行，对于同组用户可读、执行，对于其他用户可执行。

Chmod 还有一种用法就是使用包含字母和操作符表达式的字符设定法（相对权限设定），通过参数-r、-w、-x 来设定权限，这里不再详细地介绍。

- 改变文件/目录的所有权

```
chown [-R] 用户名  文件/目录
```

例如

chown TH File1

将当前目录下的文件 File 改为用户 TH 所有。

chown -R TH   Dir1

将当前目录 Dir1 改为用户 TH 所有。

（3）用户管理

- 添加/删除用户

adduser   user1，由具有 root 权限的用户添加用户 user1；

userdel   user2，由具有 root 权限的用户删除用户 user2；

- 设置用户口令

为了更好地保护用户账号的安全，Linux 允许用户随时修改自己的口令。修改口令的命令是 passwd，它将提示用户输入旧口令和新口令，之后还要求用户再次确认新口令，以避免用户无意中按错键。

（4）文件的打包和压缩

先来看一下 Linux 下打包命令。Linux 下最常用的打包程序就是 tar（tape archive-磁带存档），使用 tar 程序打出来的包都是以.tar 结尾的。Tar 命令可以为文件和目录创建档案（备份文件），也可以在档案中改变文件，或者向档案中加入新的文件。使用 tar 命令，可以把一大堆的文件和目录全部打包成一个文件，这对于备份文件或将几个文件组合成为一个文件以便于传输是非常有用的。其语法如下。

```
# tar  [选项]f   targetfile.tar   文件/目录
```

注意 选项后面的 f 是必须的，通常用来指定包的文件名。

选项说明如下。

c：创建新的档案文件。如果用户想备份一个目录或是一些文件，就要选择这个选项。例如：

```
# tar  -cf  test.tar  /home/tmp         ---
```

将/home/tmp 目录下的文件打包为 test.tar

r：增加文件到已有的包，如果发现还有一个目录或是一些文件忘记备份了，这时可以使用该选项，将还需要的目录或文件添加到包文件中，例如：

```
# tar -rf  test.tar *.jpg
```

该命令将所有的 jpg 文件添加到 test.tar 包里面去。-r 是表示增加文件的意思。

t：列出包文件的所有内容，查看已经备份了哪些文件。例如：

```
# tar  -tf  test.tar
```

x：从 tar 包文件中恢复所有文件，事实上是一个解包的过程。例如：

```
# tar  -xf  test.tar
```

k：保存已经存在的文件。例如把某个文件还原，在还原的过程中，遇到相同的文件，不会进行覆盖。

w：每一步都要求确认。

tar 命令还有一个非常重要的用法，这就是 tar 可以在打包或解包的同时调用其他的压缩程序（如 gzip、bzip2）来压缩文件。

注意 打包和压缩是两个不同的概念。

Linux 下的压缩文件主要有以下几种格式。

.Z-compress 程序的压缩格式；

.bz2-bzip2 程序的压缩格式；

.gz-gzip 程序的压缩格式；

.tar.gz-由 tar 程序打包，并且经过 gzip 程序的压缩，是 Linux 下常见的压缩文件格式；

.tar.bz2-由 tar 程序打包，并且经过 bzip2 程序的压缩。

以下就几种常用的情况进行说明。

- 调用 gzip 程序来压缩文件

gzip 是 GNU 组织开发的一个压缩程序，gzip 压缩文件的后缀是.gz，与 gzip 相对的解压程序是 gunzip。tar 中使用-z 这个参数来调用 gzip。下面举例说明。

```
# tar –czf  test.tar.gz  *.jpg
```

这条命令是将当前目录下的所有.jpg 文件打成一个 tar 包，并且将其用 gzip 程序压缩，生成一个 gzip 压缩过的包，压缩包名为 test.tar.gz，解开该压缩包的用法如下。

```
# tar –xzf  test.tar.gz
```

- 调用 bzip2 程序来压缩文件

bzip2 是 Linux 下的一个压缩能力更强的压缩程序，bzip2 压缩文件的后缀是.bz2，与 bzip2 相对应的解压程序是 bunzip2。tar 中使用-j 这个参数来调用 gzip 压缩程序。例如：

```
# tar  -cjf  test.tar.bz2  *.jpg
```

该命令是将当前目录下所有.jpg 的文件打成一个 tar 包，并且将其用 bzip2 程序压缩，生成一个 bzip2 压缩过的包，压缩包名为 test.tar.bz2，解开该压缩包的用法如下。

```
# tar  –xjf  test.tar.bz2
```

（5）rpm 软件包的安装

在使用任何操作系统的过程中，安装和卸载软件是必须的操作。Linux 中有一套软件包管理器，最初由 Red Hat 公司推出，称为 rpm（Red Hat Package Manager），它可以用来安装、查询、校验、删除、更新 rpm 格式的软件包。rpm 软件包包含可执行的二进制程序和该程序运行时所需要的文件，rpm 格式的软件包文件使用.rpm 为扩展名。与直接从源代码安装相比，软件包管理易于安装、更新和卸载软件，也易于保护配置文和跟踪已安装文件。

安装 rpm 软件包的主要格式如下。

```
#rpm –i[options] software.rpm
```

rpm 命令主要有以下参数。

- i：安装 rpm 软件包。
- t：测试安装。
- h：安装时输出 hash 记号（"#"），可以显示安装进度。
- f：忽略安装过程中的任何错误。
- U：升级安装 rpm 软件包。
- e：卸载已安装的软件包。
- V：检测软件包件是否正确安装。

以安装 develop-devel-0.9.2-2.4.5.i386.rpm 软件包为例，图 3.4 显示了它的安装过程。



图 3.4 rpm 软件包安装示例

如图 3.4 所示，系统提示的‘#’号就表示软件安装进度，当后面的的百分比数字为 100%时表示软件安装完成。

（6）源码维护基本命令

diff 命令

diff 命令是生成源代码补丁的必备工具，其命令格式如下。

diff [命令行选项] 源文件 新文件

diff 命令常用选项如下。

- -r：递归处理相应目录。
- -N：包含新文件到 patch。
- -u：输出统一格式（unified format），这种格式比缺省格式更紧凑些。
- -a：可以包含二进制文件到 patch。

通常可以使用 diff 命令加参数-ruN 来比较 2 个文件并生成一个补丁文件。这个补丁文件会列出这 2 个不同版本文件的差异。比如有 2 个文本文件：text1 和 text2，二者内容不尽相同，现在来创建补丁文件。

```
[root@localhost]# diff -ruN test1.txt  test2.txt > test.patch
```

这样就创建好了补丁文件 test.patch，补丁创建好以后需要给相应文件/程序打好补丁，这里就要用到 patch 命令。

patch [命令行选项] [patch 文件]

patch 的详细使用方法可参见 patch 的 man help，常用的命令行选项是-pn（n 是自然数），例如采用下面的指令来打好补丁。

```
[root@localhost]patch -p1 <test.patch
```

-p1 选项代表 patch 文件名左边目录的层数，考虑到顶层目录在不同的系统上可能有所不同。要使用这个选项，就要把 patch 文件放在要被打补丁的目录下，然后在这个目录中运行 path -p1 < [patchfile]命令。

（7）配置、编译、安装源码包软件

所谓源码包软件，顾名思义，就是源代码的可见的软件包，在 Linux 系统下也经常需要用到源码包软件。

大多数的源码软件包是以 tar.gz 或 tar.bz2 的形式得到的，所以在配置和编译之前需要将软件包解压缩，具体的做法已经在前面提到过。配置、编译、安装的过程大多如下所示。

```
#./configure
# make
# make install
```

./configure 用来配置软件的功能，./configure 比较重要的一个参数是--prefix，通过使用--prefix 参数，可以指定软件的安装目录；比如可以指定软件安装到/homet/tmp 目录中，可以执行如下的指令。

```
#./configure --prefix=/home/tmp
# make
# make install
```

（8）中断 Shell 命令执行的方法

在 Linux 系统下，一旦出现了 Shell 提示符，就可以键入命令名称及命令所需要的参数。Shell 将执行这些命令。如果在执行过程当中想终止命令执行，可以从键盘上按 Ctrl+C 发出中断信号来中断它。

（9）模块管理指令

Linux 内核采用模块化管理方式，这是 Linux 内核的一大特点，这也使得 Linux 整体结构非常灵活，编于精简。

- insmod（添加模块）指令

Linux 有许多功能是通过模块的方式，在需要时才载入 kernel。如此可使 kernel 较为精简，进而提高效率，以及保有较大的弹性。这些可动态加载的模块，通常是系统的设备驱动程序。加载模块采用 insmod 指令，其常用语法如下。

insmod [-fkmpsvxX] [-o<模块名称>] [模块文件]

其中的参数解释如下。

-f：不检查目前 kernel 版本与模块编译时的 kernel 版本是否一致，强制将模块载入。

-k：将模块设置为自动卸载。

-m：输出模块的载入信息。

-p：测试模块是否能正确地载入 kernel。

-s：将所有信息记录在系统记录文件中。

-v：执行时显示详细的信息。

-x：要汇出模块的外部符号。

-X：汇出模块所有的外部符号，此为预设置。

- rmmod（卸载模块）指令

Linux 把系统的许多功能编译成一个个单独的模块，待有需要时再分别加载它们，如果

不再需要这些模块的时候，就可以使用 rmmod 命令来卸载这些模块。其语法如下。

rmmod [-as] [模块名称…]

其使用参数说明如下。

-a：删除所有目前不需要的模块。

-s：把信息输出至 syslog 常驻服务，而非终端机界面。

3.1.3 编写 Shell 脚本

在 Linux 系统中，虽然有各种各样的图形化接口工具，但是 Shell 仍然是一个非常灵活的工具。Shell 不仅仅是命令的执行，而且是一种编程语言，它提供了定义变量和参数的手段以及丰富的程序控制结构。由于 Shell 特别擅长系统管理任务，尤其适合那些易用性、可维护性和便携性比效率更重要的任务，所以用户可以通过使用 Shell 使大量的任务自动化，就像使用 DOS 操作系统的过程当中，会执行一些重复性的命令。因此常将这些大量的重复性命令写成批处理命令，通过执行这个批处理命令来代替执行重复性的命令。在 Linux 系统中也有类似的批处理命令，被称作是 Shell 脚本（Script）。前面已经提到 Shell 也是一种解释性的语言，而解释执行性的语言的与编译型语言（如 C 语言）的最大不同就在于它们编写起来很方便，也很快捷，可以说，使用 Shell 脚本来完成一些特定的常用的任务是一个不错的选择。

1．建立脚本

编辑 Shell 脚本文件使用 Linux 下的普通编辑器如 vi、Emacs 等即可。Linux 下的 Shell 默认采用 Bash，所以本书也主要以 Bash 脚本为例介绍，在建立 Shell 脚本程序的开始首先应指明使用哪种 Shell 来解释所写的脚本，一般来说 Bash 脚本以"#!"开头（文件的首行），而"#!"后面同时要将所使用 Shell 的路径明确指出，比如 Bourne Shell 的路径为/bin/sh，而 C Shell 的路径则为/bin/csh，Linux 下默认采用 Bash，所以本书也主要以 Bash 脚本为例介绍，下面的语句就是指定 Bash 来解释脚本。

```
#! /bin/sh
```

该语句说明该脚本文件是一个 Bash 程序，需要由/bin 目录下的 Bash 程序来解释执行。除了在脚本内指定所使用的 Shell 类型以外，使用过程中也可以在命令行中强制指定。比如想用 C Shell 执行某个脚本，就可以使用以下命令。

```
# csh  Myscript
```

为了增加程序的可读性，Shell 脚本语句也可以像高级语言那样加注释，在 Bash 脚本程序中从“#”号开始到行尾的部分均被看作是程序的注释语句。

2．Shell 变量

Shell 编程中可以使用变量，这充分体现了它的灵活性。对 Shell 来讲，所有变量的取值都是一个字串。Shell 脚本中主要有以下几种变量：系统变量，环境变量，用户变量。其中用户变量在编程过程中使用频繁；系统变量在对参数判断和命令返回值判断会使用；环境变量

主要是在程序运行的时候需要设置。此外，Shell 脚本的执行并不需要编译，所以也就不需用检查脚本中变量的类型，因此在 Shell 脚本中使用变量不必像高级语言那样事先对变量进行定义。

- Shell 系统变量

以下是一些常用到的 Shell 系统变量及其含义。

$ # ：　　保存程序命令行参数的数目。

$? ：　　保存前一个命令的返回值。

注意　在 Linux 中，命令退出状态为 0 表示该命令正确执行，任何非 0 值表示命令出错。

$ 0 ：　　当前程序名。

$ * ：　　以（"$1 $2…"）的形式保存所有输入的命令行参数。

$ @ ：　　以（"$1""$2"…）的形式保存所有输入的命令行参数。

$ n ：　　$1 为命令行的第一个参数，$1 为命令行的第二个参数，依次类推。

举一个针对以上系统变量使用的例子，使用 vi 编辑一个脚本文件，文件名为 Example Script，其内容如下。

```
#! /bin/sh
# Script name: Example Script
echo "The No. of parameter is:  $#";
echo "The script name is: $0";
echo "The parameters in the script are: $*";
```

在命令行中执行该脚本程序：

./Example Script　Hello Linux

命令行中的 Hello Linux 是其参数，该程序执行结果如下。

```
The No. of parameter is: 2
The script name is: ./ Example Script
The parameters in the script are: Hello Linux
```

- Shell 环境变量

Shell 环境变量是所有 Shell 程序都会接受的参数。Shell 程序运行时，都会接收一组变量，这组变量就是环境变量，常用的 Shell 环境变量如下。

PATH：决定了 Shell 将到哪些目录中寻找命令或程序。

HOME：当前用户主目录的完全路径名。

HISTSIZE：历史记录数。

LOGNAME：当前用户的登录名。

HOSTNAME：指主机的名称。

SHELL：Shell 路径名。

LANGUGE：语言相关的环境变量，多语言可以修改此环境变量。

MAIL：当前用户的邮件存放目录。

PS1：主提示符，对于 root 用户是#，对于普通用户是$。

PS2：辅助提示符，默认是“>”。

TERM：终端的类型。

PWD：当前工作目录的绝对路径名。

- Shell 用户变量

Shell 用户变量是最常使用的变量，可以使用任何不包含空格字符的字串来当做变量名称，在 Linux 支持的所有 Shell 中，都可以用赋值符号（=）为变量赋值，在使用 Shell 用户变量的时候，通常是按照下面的语法规则来定义用户变量。

变量名=变量值

例如：

```
A=9
B="Hello World"
```

注意 在定义变量时，变量名前不应加符号$，等号两边一定不能留空格。

变量的引用，要在变量前加$，例如：

```
S="string"
echo $S
```

下面举一个非常简单的例子来说明。

```
#! /bin/bash
# This is a example
SR="Hello World"
echo $STR
```

上面的例子很简单，定义了一个变量 SR，并且赋值给 SR，然后在终端输出 SR 的值。

3．流程控制

同传统的编程语言一样，Shell 提供了很多特性，如数据变量、参数传递、判断、流程控制、数据输入和输出、子程序及以中断处理等。

（1）条件语句

同其他高级语言程序一样，复杂的 Shell 程序中经常使用到分支和循环控制结构，主要有 2 种不同形式的条件语句：if 语句和 case 语句。

- if 语句

if 语句的语法格式如下。

```
if   [expression]
then
```

```
commands1      // expression 为 True 时的动作
else
commands2     // expression 为 False 时的动作
fi
.
.
```

- case 语句

case 语句的语法格式如下。

```
case 字符串 in
 模式 1） command;;
 模式 2） command;;
……
esac
```

case 语句是多分支语句，它按“）”左边的模式对字符串值的匹配来执行相应的命令，匹配总是由上而下地进行，总是执行首先匹配到的模式对应的命令表，如果模式中的每个都匹配不到，则什么也不执行，所以一般会在最后，放一个*)，代表以上都不匹配的任意字符串。";;"表示该模式对应的命令部分程序。

（2）循环语句

- while 循环语句

在 while 循环语句中，当某一条件为真时，执行指定的命令。语句的结构如下。

```
while expression
do
command
……
done
```

- for 循环语句

for 循环语句对一个变量的可能的值都执行一个命令序列。赋给变量的几个数值既可以在程序内以数值列表的形式提供，也可以在程序以外以位置参数的形式提供。for 循环语句的一般格式如下。

```
for   变量名 [in 列表]
do
    command1
    command2
   ……
done
```

4．Shell 脚本的执行

Shell 脚本是以文本方式存储的，而非二进制文件。所以 Shell 脚本必须在 Linux 系统的 Shell 下解释执行。如果已经写好 Shell 脚本，运行该脚本可以有以下的几种方法。

（1）设置好脚本的执行权限之后再执行脚本

可以使用下列方式设置脚本的执行权限。

- chmod u+x Scriptname 只有自己可以执行，其他人不能执行；
- chmod ug+x Scriptname 只有自己以及同一群可以执行，其他人不能执行；
- chmod +x Scriptname 所有人都可以执行。

设置好执行权限之后就可以执行脚本程序了，例如，编辑好一个脚本程序 MyScript 之后，可按下面的方式来执行。

```
[localhost@zxq]# chmod  +x  MyScript
[localhost@zxq]# ./Myscript
```

（2）使用 Bash 内部指令"source"

例如下面的执行过程：

```
[localhost@zxq]# source  Myscript
```

（3）直接使用 sh 命令来执行

例如：

```
[localhost@zxq]# sh  Myscript
```

注意 后面的两种情况不必设置权限即可执行。

3.1.4　正则表达式

正则表达式源于人类神经系统如何工作的早期研究。在 19 世纪 60 年代，一位叫 Stephen Kleene 的数学家发表了一篇标题为“神经网事件的表示法”的论文，正式引入了正则表达式的概念。正则表达式就是用来描述他称为“正则集的代数”的表达式，因此采用“正则表达式”这个术语，此后，正则表达式的第一个实用应用程序就是 Unix 中的 qed 编辑器。

在 Shell 编程中经常会用到正则表达式（regular expression），简单地讲，正则表达式是一种可以用于模式匹配和替换的有效工具。正则表达式描述了一种字符串匹配的模式，可以用来检查一个串是否含有某种子串、将匹配的子串做替换或者从某个串中取出符合某个条件的子串等。使用 Shell 时，从一个文件中抽取多于一个字符串有时会很不方便，而使用正则表达式可以方便快捷地解决这一问题。

正则表达式由普通字符（例如字符 a 到 z）以及特殊字符（称为特殊字符）组成特定文字模式。当从一个文件或命令中抽取或者过滤文本时，使用正则表达式可以简化命令中的匹配表达。Linux 系统自带的所有文本过滤工具在某种模式下都支持正则表达式，正则表达式可以匹配行首与行尾、数据集、字母和数字以及一定范围内的字符串集合，在进行匹配时，

正则表达式有一组基本特殊字符，其基本的特殊字符及其含义如表 3.1 所示。

表 3.1 正则表达式特殊字符及其含义

特 殊 字 符	代 表 含 义
^	只匹配行首
$	只匹配行尾
*	单字符后跟*将匹配 0 个或者多个此字符
[ ]	匹配[ ]内的字符，可以是单个字符也可以是字符序列
\	转义字符，用来屏蔽一个字符的特殊含义
.	用来匹配任意的单字符
Pattern\{\n}	用来匹配 pattern 在前面出现的次数，n 即为次数
Pattern\{n,\}	用来匹配前面 pattern 出现的次数，次数最少为 n
Pattern\{n,m\}	用来匹配前面 pattern 出现的次数，次数在 n 和 m 之间

下面举几个简单的例子来说明。

1．行首和行尾的匹配

在 Bash 中使用正则表达式时，可以使用^和$来分别匹配行首和行尾的字符或字符串，比如下面的正则表达式。

^....abc..

该表达式的含义是在每行开始任意匹配 4 个字符，之后必须是字符 abc，行尾匹配任意的 3 个字符，那么该表达式与下面各个字符串的匹配结果如下。

Gyftabc12345　　不匹配（行尾不匹配）

7853abcpoi　　匹配

85fabc0k8　　不匹配（行首不匹配）

2．[ ]和指定次数的匹配

括号[ ]用来匹配特定字符串和字符串集合，可以用逗号将要匹配的不同字符串分开，用“-”符号表示匹配字符串的范围，例如，想要匹配任意的字母和数字，可以使用下面的正则表达式。

[A-Z，a-z，0-9]

*号可以匹配单字符 0 次或多次，例如下面的字符串都可以与表达式 Des*k 匹配。

Desk

Dessk

Dessskl

使用*可匹配所有匹配结果任意次，如果要指定匹配的次数，就应使用\ { \ }用法，使用有 3 种模式。

- pattern\{n\} 匹配模式出现 n 次。

- pattern\{n,\} 匹配模式出现至少 n 次。
- pattern\{n, m} 匹配模式出现次数在 n 到 m 次之间，n, m 为 0～255 中的任意整数。

例如表达式 G\{2\}H、G\{2,}\H、G\{2,3}的匹配结果分别如下。

GGH

GG（...,多个 G）H

GGH,GGGH

3．使用反斜杠\来屏蔽一个特殊字符的含义

有时在进行文本过滤或抽取的时候，所要匹配的字符本身就是特殊字符，但并没有特殊的含义，为了将二者区分开来，就需要用到反斜杠来转义该字符（也称转义符）。比如要匹配包含“*”的字符串，而“*”是一个特殊字符，因此需要屏蔽它的特殊含义，就可以如下操作。

*

这样的表示方式就认为*是一个特殊的字符，再比如要匹配包含“^”的语句，可以如下表示。

\^

反斜杠\将^的特殊含义屏蔽，在这里只是代表一个普通字符^。

构造正则表达式的方法和创建数学表达式的方法一样，采用多种元字符与操作符将一些基本的表达式组合成为功能更复杂的正则表达式，其组成元素可以是单个的字符、字符集、字符或数字的范围、字符间的选择或者所有这些元素的任意组合。表 3.2 是常用到的一些正则表达式。

表 3.2　　常用正则表达式特及其含义

表　达　式	代 表 含 义
^	只匹配行首
$	只匹配行尾
^[STR]	匹配以 STR 作为行的开头
[Ss]igna[lL]	匹配单词 s i g n a l、Signal、s i g n a L、SignaL
^USER$	匹配只包含 USER 的行
^d..x..x..x	匹配对用户、用户组和其他用户组成员都有可执行权限的目录
[.*0]	用来匹配 0 之前或之后加任意字符
[^$]	用来匹配空行
[^.*$]	用来匹配行中任意字符串
[a-z][a-z]*	至少一个小写字母
[^0-0A-Za-z]	匹配非数字或字母（大小写均可）
[i I] [n N]	匹配大写或小写的 i/n
\ .	匹配带句点的行
[0 0 0 *]	匹配 0 0 0 或更多个 0

^.*	匹配只有一个字符的行

3.1.5 程序编辑器

编辑器是系统的重要工具之一，在各种操作系统中，编辑器都是必不可少的部件。Linux 系统提供了一个完整的编辑器家族系列，如 Ed、Ex、Vi 和 Emacs 等，按功能它们可以分为两大类。

- 行编辑器（如 Ed、Ex）
- 全屏幕编辑器（如 Vi、Emacs）

行编辑器每次只能对一行进行操作，使用起来不是很方便。而全屏幕编辑器可以对整个屏幕进行编辑，用户编辑的文件直接显示在屏幕上，修改的结果可以立即看出来，克服了行编辑方式存在的一些缺点，便于用户学习和使用。Vi（Visual Interface）和 Emacs（Editing with MACroS）是 Linux 下主要的 2 个编辑器，下面的内容主要就 Vi 的使用做详细的介绍。

Vi 编辑器最初是由 Sun Microsystems 公司的 Bill Joy 在 1976 年开发的。一开始 Bill 开发了 Ex 编辑器，后来开发了 Vi 作为 Ex 的 visual interface，也就是说 Vi 允许一次能看到一屏的文本而非一行，Vi 也因此得名。随之技术的不断进步，基于 Vi 的各种变种版本不断出现，其中，移植特性最好，使用最广泛的当属 Vim 编辑器，相比早期的 Vi，Vim 编辑器增加的一项最重要的功能便是多级撤销，Vi 只支持一级撤销。

目前，Vi/Vim 已经是 Linux 下用的最普遍的文本处理器之一，Vi 也是 Linux 下的第一个全屏幕交互式编辑程序，使用非常普遍，Vi 没有菜单，只有命令，且命令繁多，但是一旦掌握了 Vi 的用法，就可以体会到它的强大功能。它可以执行输出、删除、查找、替换、块操作等众多文本操作，而且用户可以根据自己的需要对其进行定制，这是其他编辑程序所没有的。在终端下输入 Vim 命令就可以看到 Vi 的界面了，如图 3.5 所示。

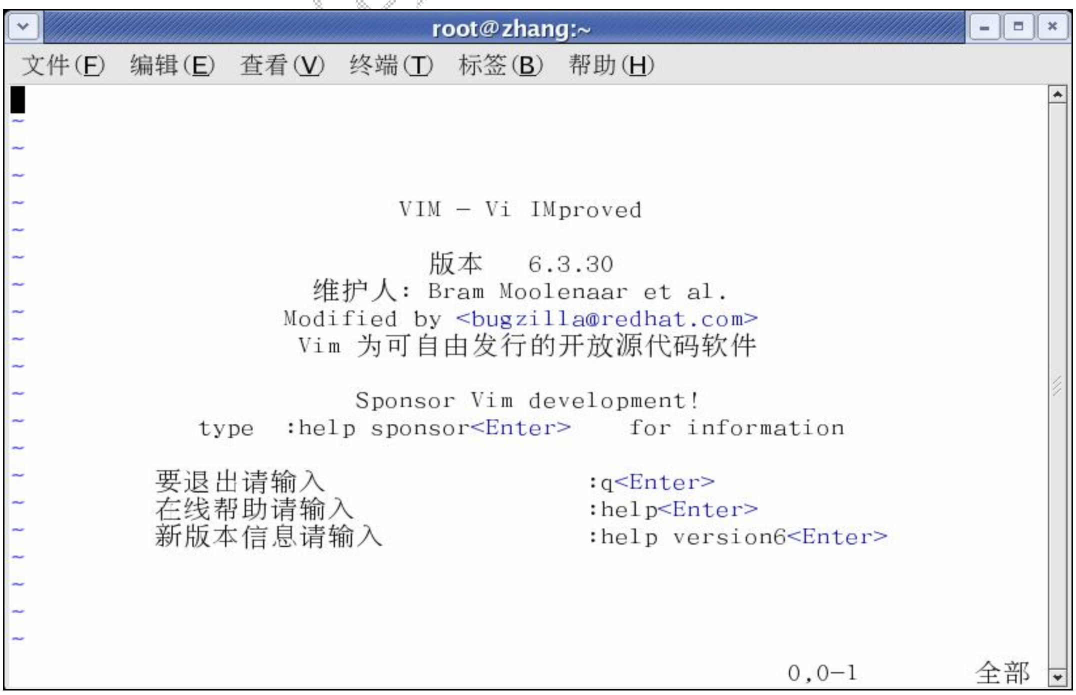


图 3.5 Vi 界面

Vi 有 3 种基本工作模式：指令行模式、文本输入模式、末行模式，它们的相互关系如图 3.6 所示。

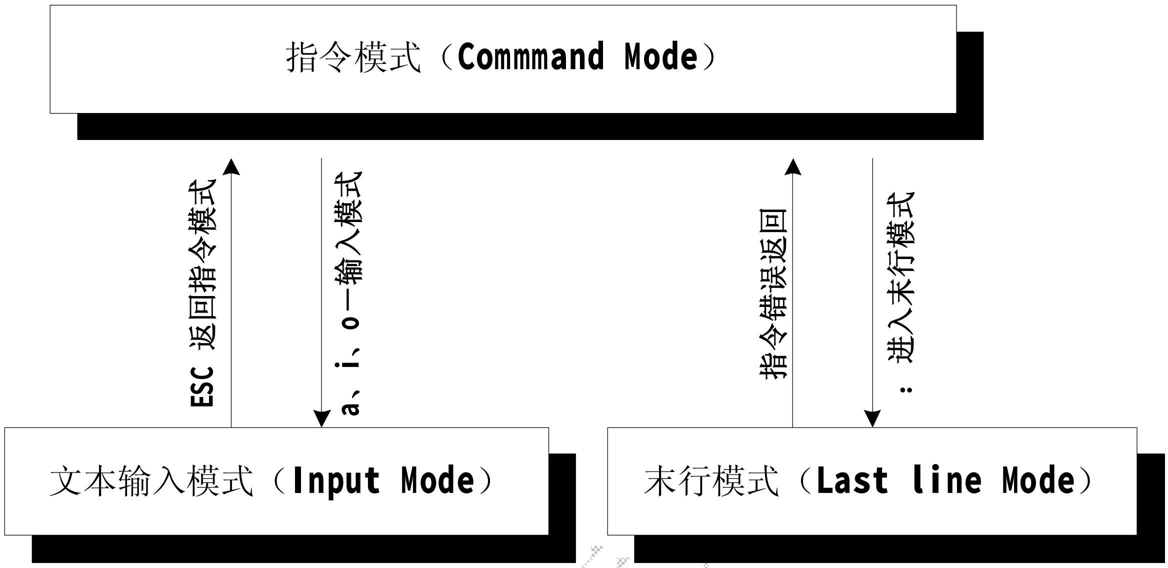


图 3.6 Vi 的模式切换关系

下面分别来介绍这 3 种模式。

1．指令模式（command mode）

指令模式主要使用方向键移动光标位置以进行文字的编辑，在输入模式下按【Esc】键或是在末行模式输入了错误命令，都会回到指令模式，表 3.3 列出了其常用操作命令及含义。

表 3.3 vi 指令模式命令及其含义

操 作 命 令	实 现 功 能
0	光标移至行首
h	光标左移一格
l	光标右移一格
j	光标向下移一行
k	光标向上移一行
$ + A	将光标移到该行最后
PageDn	向下滚动一页
PageUp	向上滚动一页
d+方向键	删除文字
dd	删除整行

pp	整行复制
r	修改光标所在字符
S	删除光标所在的列，并进入输入模式

2．文本输入模式

在 vim 下编辑文字，不能直接插入、替代或删除文字，而必须先进入输入模式。要进入输入模式，可以在指令模式下按【a/A】键、【i/I】键或【o/O】键，它们的命令及其含义如表 3.4 所示。

表 3.4　　文本输入模式命令及其含义

操 作 命 令	实 现 功 能
a	在光标后开始插入
A	在行尾开始插入
i	从光标所在位置前面开始插入
I	从光标所在列的第一个非空白字元前面开始插入
o	在光标所在列下新增一列并进入输入模式
O	在光标所在列上方新增一列并进入输入模式
Esc	返回命令行模式

注意　结束文本输入模式必须用【Esc】键。

3．末行模式

末行模式主要用来进行一些文字编辑辅助功能，比如字串搜寻、替代、保存文件等，表 3.5 介绍一些常用的命令。

表 3.5　　末行模式命令及其含义

操 作 命 令	实 现 功 能
：q	结束 Vi 程序，如果文件有过修改，先保存文件
：q!	强制退出 Vi 程序
：wq	保存修改并退出程序
：set nu	设置行号

大多数时候，可用命令如：Vi filename 来打开文件 filename，Vim 以编辑或打开某个文件。下面以编辑一个简单脚本程序为例介绍 Vi 的简单使用方法，其主要流程如下。

- 在终端输入命令用 Vi 建立文件（可以是文本文件、C\C++程序等）

```
# vi  Script_edit
```

输入该命令之后就进入了 Vi 的编辑界面，如图 3.7 所示。



图 3.7 Vi 编辑界面

此时的 Vi 是指令模式，输入“：set nu”来设置行号，此时属于末行模式，末行模式不能直接切换到文本输入模式，需要先切换到指令模式，按【Esc】键进入指令模式。



- 输入“i”进入输入模式。

在指令模式下输入“i”进入文本输入模式，并编辑文本内容，如图 3.8 所示。

- 保存、修改编辑内容并退出 Vi 程序。

在输入模式下编辑并修改相应内容，编辑好之后需要再返回到指令模式（Esc），之后输入“：wq”就可以保存并且退出刚才的编辑程序了。

3.2 Makefile

3.2.1 GNU make

GNU make 最初是 Unix 系统下的一个工具，设计之初是为了维护 C 程序文件不必要的重新编译，它是一个自动生成和维护目标程序的工具。在使用 GNU 的编译工具进行开发时，经常要用到 GNU make 工具。使用 make 工具，我们可以将大型的开发项目分解成为多个更易于管理的模块，对于一个包括几百个源文件的应用程序，使用 make 和 Makefile 工具就可

以高效的处理各个源文件之间复杂的相互关系，进而取代了复杂的命令行操作，也大大提高了应用程序的开发效率，可以想到的是如果一个工程具有上百个源文件时，但是采用命令行逐个编译那将是多么大的工作量。

使用 make 工具管理具有多个源文件的工程，其优势是显而易见的，举一个简单的例子，如果多个源文件中的某个文件被修改，而有其他多个源文件依赖该文件，采用手工编译的方法需要对所有与该文件有关的源文件进行重新编译，这显然是一件费时费力的事情，而如果采用 make 工具则可以避免这种繁杂的重复编译工作，大大地提高了工作效率。

make 是一个解释 Makefile 文件中指令的命令工具，其最基本的功能就是通过 Makefile 文件来描述源程序之间的相互关系并自动维护编译工作，它会告知系统以何种方式编译和链接程序。一旦正确完成 Makefile 文件，剩下的工作就只是在 Linux 终端下输入 make 这样的一个命令，就可以自动完成所有编译任务，并且生成目标程序。通常状况之下 GNU make 的工作流程如下。

① 查找当前目录下的 Makefile 文件
② 初始化文件中的变量
③ 分析 Makefile 中的所有规则
④ 为所有的目标文件创建依赖关系
⑤ 根据依赖关系，决定哪些目标文件要重新生成
⑥ 执行生成命令

为了比较形象地说明 make 工具的工作原理，举一个简单的例子来介绍。假定一个项目中有以下一些文件。

- 源程序：Main.c、test1.c、test.c。
- 包含的头文件：head1.h、head2.h、head3.h。
- 由源程序和头文件编译生成的目标文件：Main.o、test1.o、test2.o。
- 由目标文件链接生成的可执行文件：test。

这些不同组成部分的相互依赖关系如图 3.9 所示。

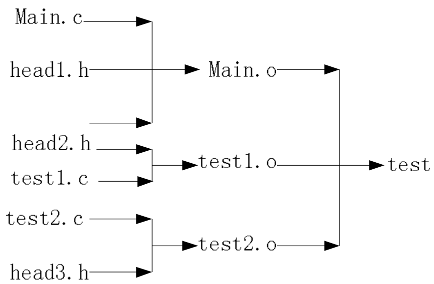


图 3.9　依赖关系

在该项目的所有文件当中，目标文件 Main.o 的依赖文件是 Main.c、head1.h、head2.h；test1.o 的依赖文件是 head2.h、test1.c；目标文件 test2.o 的依赖文件是 head3.h、test2.c；最终的可执行文件的依赖文件是 Main.o、test1.o 和 test2.o。执行 make 命令时，会首先处理 test 程序的所有依赖文件（.o 文件）的更新规则，对于.o 文件，会检查每个依赖程序（.c 和.h 文件）是否有更新，判断有无更新的依据主要看依赖文件的建立时间是否比所生成的目标文件要晚，如果是，那么会按规则重新编译生成相应的目标文件，接下来对于最终的可执行程序，同样会检查其依赖文件（.o 文件）是否有更新，如果有任何一个目标文件要比最终可执行的目标程序新，则重新链接生成新的可执行程序，所以，make 工具管理项目的过程是从最底层开始的，是一个逆序遍历的过程。从以上的说明就能够比较容易理解使用 make 工具的优势了，事实上，任何一个源文件的改变都会导致重新编译、链接生成可执行程序，使用者不必关心哪个程序改变、或者依赖哪个文件，make 工具会自动完成程序的重新编译和链接工作。

执行 make 命令时，只需在 Makefile 文件所在目录输入 make 指令即可，事实上，make 命令本身可带有这样的一些参数：【选项】、【宏定义】、【目标文件】。其标准形式如下。

Make [选项] [宏定义] [目标文件]

Make 命令的一些常用选项及其含义如下。

- -f file：指定 Makefile 的文件名。
- -n：打印出所有执行命令，但事实上并不执行这些命令。
- -s：在执行时不打印命令名。
- -w：如果在 make 执行时要改变目录，则打印当前的执行目录。
- -d：打印调试信息。
- -I<dirname>：指定所用 Makefile 所在的目录。
- -h：help 文档，显示 Makefile 的 help 信息。

举例来讲，在使用 make 工具的时候，习惯把 makefile 文件命名为 Makefile，当然也可以采用其他的名字来命名 makefile 文件，如果要使用其他文件作为 Makefile，则可利用带-f 选项的 make 命令来指定 Makefile 文件。

```
# make –f  Makefilename
```

参数【目标文件】对于 make 命令来说也是一个可选项，如果在执行 make 命令时带有该参数，可以输入如下的命令。

```
# make  target
```

target 是用户 Makefile 文件中定义的目标文件之一，如果省略参数 target，make 就将生成 Makefile 文件中定义的第一个目标文件。因此，常见的用法就是经常把用户最终想要的目标文件（可执行程序）放在 Makefile 文件中首要的位置，这样用户直接执行 make 命令即可。

3.2.2　Makefile 规则语法

简单地讲，Makefile 的作用就是让编译器知道要编译一个文件需要依赖哪些文件，同时当那些依赖文件有了改变，编译器会自动的发现最终的生成文件已经过时，而重新编译相应的模块。Makefile 的内容规定了整个工程的编译规则。一个工程中的许多源文件按其类型、功能、模块可能分别被放在不同的目录中，Makefile 定义了一系列的规则来指定，比如哪些文件是有依赖性的，哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译。

Makefile 有其自身特定的编写格式并且遵循一定的语法规则。

```
#注释
目标文件：依赖文件列表
...............
<Tab>命令列表
...............
```

格式的说明如下。

- 注释：和 Shell 脚本一样，Makefile 语句行的注释采用“#”符号。

- 目标：目标文件的列表，通常指程序编译过程中生成的目标文件（.o 文件）或最终的可执行程序，有时也可以是要执行的动作，如“clean”这样的目标。
- 依赖文件：目标文件所依赖的文件，一个目标文件可以依赖一个或多个文件。
- “：”符号，分隔符，介于目标文件和依赖文件之间。
- 命令列表：make 程序执行的动作，也是创建目标文件的命令，一个规则可以有多条命令，每一行只能有一条命令。

注意 每一个命令行必须以[Tab]键开始，[Tab]告诉 make 程序该行是一个命令行，make 按照命令完成相应的动作。

从上面的分析可以看出，Makefile 文件的规则其实主要有两个方面，一个是说明文件之间的依赖关系，另一个是告诉 make 工具如何生成目标文件的命令。下面是一个简单的 makefile 文件例子。

```
#Makefile Example
test: main.o test1.o test2.o
	gcc –o test main.o test1.o test2.o
main.o: main.c head1.h head2.h
	gcc –c main.c
test1.o: test1.c  head2.h
	gcc –c test1.c
test2.o: test2.c head3.h
	gcc –c test2.c
install:
	cp  test  /home/tmp
clean:
	rm –f *.o
```

在这个 makefile 文件中，目标文件（target）即为：最终的可执行文件 test 和中间目标文件 main.o、test1.o、test2.o，每个目标文件和它的依赖文件中间用“：”隔开，依赖文件的列表之间用空格隔开。每一个.o 文件都有一组依赖文件，而这些.o 文件又是最终的可执行文件 test 的依赖文件。依赖关系实质上就是说明了目标文件是由哪些文件生成的。

在定义好依赖关系后，在命令列表中定义了如何生成目标文件的命令，命令行以 Tab 键开始。Make 工具会比较目标文件和其依赖文件的创建或修改日期，如果所依赖文件比目标文件要新，或者目标文件不存在的话，那么，make 就会执行命令行列表中的命令来生成目标文件。

3.2.3 Makefile 文件中变量的使用

Makefile 文件中除了一系列的规则，对于变量的使用也是一个很重要的内容。Linux 下的 Makefile 文件中可能会使用很多的变量，定义一个变量（也常称为宏定义），只要在一行的开始定义这个变量（一般使用大写，而且放在 Makefile 文件的顶部来定义），后面跟一个=

号，=号后面即为设定的变量值。如果要引用该变量，用一个$符号来引用变量，变量名需要放在$后的()里。

make 工具还有一些特殊的内部变量，它们根据每一个规则内容定义。

- $@：指代当前规则下的目标文件列表。
- $<：指代依赖文件列表中的第一个依赖文件。
- $^：指代依赖文件列表中所有依赖文件。
- $?：指代依赖文件列表中新于对应目标文件的文件列表。

变量的定义可以简化 makefile 的书写，方便对程序的维护。例如前面的 Makefile 例程就可以如下书写。

```
#Makefile Example
OBJ=main.o test1.o test2.o
CC=gcc
test: $（OBJ）
      $（CC） -o test  $（OBJ）
main.o: main.c head1.h head2.h
     $（CC）-c main.c
test1.o: test1.c  head2.h
     $（CC） -c test1.c
test2.o: test2.c head3.h
     $（CC） -c test2.c
install:
     cp  test  /home/tmp
clean:
     rm  -f  *.o
```

从上面修改的例子可以看到，引入了变量 OBJ 和 CC，这样可以简化 makefile 文件的编写，增加了文件的可读性，而且便于修改。举个例子来说，假定项目文件中还需要加入另外一个新的目标文件 test3.o，那么在该 Makefile 中有两处需要分别添加 test3.o；而如果使用变量的话只需在 OBJ 变量的列表中添加一次即可，这对于更复杂的 Makefile 程序来说，会是一个不小的工作量，但是，这样可以降低因为编辑过程中的疏漏而导致出错的可能。

一般来说，Makefile 文件中变量的应用主要有以下几个方面。

1. 代表一个文件列表

Makefile 文件中的变量常常存储一些目标文件甚至是目标文件的依赖文件，引用这些文件的时候引用存储这些文件的变量即可，这给 Makefile 编写和维护者带来了很大的方便。

2. 代表编译命令选项

当所有编译命令都带有相同编译选项时（比如-Wall -O2 等），可以将该编译选项赋给一个变量，这样方便了引用。同时，如果改变编译选项的时候，只需改变该变量值即可，而不

必在每处用到编译选项的地方都做改动。

在上面的 Makefile 例子中，还定义了一个伪目标 clean，它规定了 make 应该执行的命令，即删除所有编译过程中产生的中间目标文件。当 make 处理到伪目标 clean 时，会先查看其对应的依赖对象。由于伪目标 clean 没有任何依赖文件，所以 make 命令会认为该目标是最新的而不会执行任何操作。为了编译这个目标体，必须手工执行如下命令。

```
# make  clean
```

此时，系统会有提示信息：

rm -f *.o

另一个经常用到的伪目标是 install。它通常是将编译完成的可执行文件或程序运行所需的其他文件拷贝到指定的安装目录中，并设置相应的保护。例如在上面的例子中，如果用户执行命令：

```
# make  install
```

系统会有提示信息：

cp test1 /home/tmp

也即是将可执行程序 test1 拷贝到系统/home/tmp 下。事实上，许多应用程序的 Makefile 文件也正是这样写的，这样便于程序在正确编译后可以被安装到正确的目录。

3.3　二进制代码工具的使用

3.3.1　GNU Binutils 工具介绍

在 Linux 下建立嵌入式交叉编译环境要用到一系列的工具链（tool-chain），主要有比如 GNU Binutils、Gcc、Glibc、Gdb 等，它们都属于 GNU 的工具集。其中，GNU Binutils 是一套用来构造和使用二进制所需的工具集。建立嵌入式交叉编译环境，Binutils 工具包是必不可少的，而且 Binutils 与 GNU 的 C 编译器 gcc 是紧密相集成的，没有 binutils，gcc 也不能正常工作的。Binutils 的官方下载地址是：ftp://ftp.gnu.org/gnu/binutils/，在这里可以下载到不同版本的 Binutils 工具包。目前比较新的版本是 Binutils-2.16.1。GNU Binutils 工具集里主要有以下一系列的部件。

- as　GNU 的汇编器

作为 GNU Binutils 工具集中最重要的工具之一。as 工具主要用来将汇编语言编写的源程序转换成二进制形式的目标代码。Linux 平台的标准汇编器是 GAS，它是 GNU GCC 编译器所依赖的后台汇编工具，通常包含在 Binutils 软件包中。

- ld　GNU 的链接器

同 as 一样，ld 也是 GNU Binutils 工具集中重要的工具，Linux 使用 ld 作为标准的链接程序，由汇编器产生的目标代码是不能直接在计算机上运行的，它必须经过链接器的处理才能生成可执行代码，链接是创建一个可执行程序的最后一个步骤，ld 可以将多个目标文件链接成为可执行程序，同时指定了程序在运行时是如何执行的。

- add2line　将地址转换成文件名或行号对，以便调试程序
- ar　从文件中创建、修改、扩展文件
- gasp　汇编宏处理器
- nm　从目标代码文件中列举所有变量（包括变量值和变量类型），如果没有指定目标文件，则默认是 a.out 文件
- objcopy　objcopy 工具使用 GNU BSD 库，它可以把目标文件的内容从一种文件格式复制到另一种格式的目标文件中

在默认的情况下，GNU 编译器生成的目标文件格式为 elf 格式，elf 文件由若干段（section）组成，如果不作特殊指明，由 C 源程序生成的目标代码中包含如下段：.text（正文段）包含程序的指令代码；.data（数据段）包含固定的数据，如常量、字符串；.bss（未初始化数据段）包含未初始化的变量、数组等。C++源程序生成的目标代码中还包括.fini（析构函数代码）和.init（构造函数代码）等。链接生成的 elf 格式文件还不能直接下载到目标平台来运行执行，需要通过 objcopy 工具生成最终的二进制文件。连接器的任务就是将多个目标文件的.text、.data 和.bss 等段连接在一起，而连接脚本文件是告诉连接器从什么地址开始放置这些段。

- add2line　把程序地址转换为文件名和行号

在命令行中带一个地址和一个可执行文件名，它就会使用这个可执行文件的调试信息指出在给出的地址上是哪个文件以及行号。

- objdump　显示目标文件信息

objdump 工具可以反编译二进制文件，也可以对对象文件进行反汇编，并查看机器代码。

- readelf　显示 elf 文件信息

readelf 命令可以显示符号、段信息、二进制文件格式的信息等，这在分析编译器如何从源代码创建二进制文件时非常有用。

- ranlib　生成索引以加快对归档文件的访问，并将其保存到这个归档文件中

在索引中列出了归档文件各成员所定义的可重分配目标文件。

- size　列出目标模块或文件的代码尺寸

size 命令可以列出目标文件每一段的大小以及总体的大小。默认情况下，对于每个目标文件或者一个归档文件中的每个模块只产生一行输出。

- strings　打印可打印的目标代码字符（至少 4 个字符），打印字符多少可以控制

对于其他格式的文件，打印字符串。打印某个文件的可打印字符串，这些字符串最少 4 个字符长，也可以使用选项“-n”设置字符串的最小长度。默认情况下，它只打印目标文件初始化和可加载段中的可打印字符；对于其他类型的文件它打印整个文件的可打印字符，这个程序对于了解非文本文件的内容很有帮助。

- strip　放弃所有符号连接

删除目标文件中的全部或者特定符号。

- c++filt　链接器 ld 使用该命令可以过滤 C++符号和 Java 符号，防止重载函数冲突
- gprof　显示程序调用段的各种数据

3.3.2　Binutils 工具软件使用

就 Binutils 工具软件的使用问题，以下以 Binutils 工具包中两个常用的工具的使用进行简单的说明。

1．汇编器

Linux 平台的标准汇编器是 GAS，它是 GCC 所依赖的后台汇编工具，通常包含在 binutils 软件包中。GAS 使用标准的 AT&T 汇编语法，可以用来汇编用 AT&T 格式编写的程序，例如可以这样来编译用汇编语言编写的源程序 test.s。

```
[root@localhost]# as  -o test.o  test.s
```

2．链接器

GNU 链接器使用一个命令语言脚本来控制链接过程。默认情况下，ld 是由一组内部命令进行控制的，这些命令可以进行扩展或覆盖。强调可移植性和灵活性在 GCC 的功能中是非常明显的一条，它可以为很多不同的编译环境生成链接脚本，并向 ld 传递定制过的链接脚本，而不用手工进行干预。

需要注意的是，在 Linux 下编写应用程序（假定采用 gcc 编译器）时，gcc 编译器内置缺省的连接脚本。如果采用缺省脚本，则生成的目标代码需要操作系统才能加载运行。

就像前面讲到的，由汇编器产生的目标代码是不能直接在计算机上运行的，它必须经过链接器的处理才能生成可执行代码。Linux 使用 ld 作为标准的链接程序，比如我们可以用下面的方法来链接上述编译的程序。

```
[root@localhost]# ld -s –o  test  test.o
```

这样就生成了最终的可执行程序 test。

3.4　编译器 GCC 的使用

3.4.1　GCC 编译器介绍

GCC 是 GNU 项目的编译器组件之一，也是 GNU 软件产品家族具有代表性的作品。在 GCC 设计之初，仅仅是作为一个 C 语言的编译器，可是经过十多年的发展，GCC 已经不仅仅能支持 C 语言；它现在还支持 Ada 语言，C++语言，Java 语言，Objective C 语言，Pascal 语言，COBOL 语言，以及支持函数式编程和逻辑编程的 Mercury 语言，等等。而 GCC 也不再单只是 GNU C Compiler 的意思了，而是变成了 GNU Compiler Collection 也即是 GNU 编译器家族的意思了，目前已成为 Linux 下最重要的软件开发工具之一。GCC 的发展大体经历了下面的几个阶段。

① 1987 年，第一版的 GCC 发布。

② 2001.6.18，GCC3.0 正式发布。

③ 2004.4.20，GCC 3.4.0 版本发布。

④ 2005.4.22，最新版本的 GCC 4.0 发布，官方网站：http://gcc.gnu.org。

GCC 是一个交叉平台的编译器，目前支持几乎所有主流 CPU 处理器平台，它可以完成从 C、C++、objective-C 等源文件向运行在特定 CPU 硬件上的目标代码的转换，GCC 不仅功能非常强大，结构也异常灵活，便携性（portable）与跨平台支持（cross-platform support）特性是 GCC 的显著优点，目前，GCC 编译器所能够支持的源程序的格式如表 3.6 所示。

表 3.6 GCC 所支持的源程序格式

后 缀 格 式	说 明
.c	C 语言源程序
.a	由目标文件构成的档案库文件
.C;；cc；.cxx	C++源程序
.h	源程序包含的头文件
.i	经过预处理的 C 程序
.ii	经过预处理的 C++程序
.m	Objective-C 源程序
.o	编译后的目标文件
.s	汇编语言源程序
.S	经过预编译的汇编程序

GCC 是一组编译工具的总称，其软件包里包含众多的工具，按其类型，主要有以下的分类。

① C 编译器 cc, cc1, cc1plus, gcc

② C++编译器 c++, cc1plus, g++

③ 源码预处理程序 cpp, cpp0

④ 库文件 libgcc.a, libgcc_eh.a, libgcc_s.so, libiberty.a, libstdc++.[a,so], libsupc++.a

用 GCC 编译程序生成可执行文件有时候看起来似乎仅通过编译一步就完成了，但事实上，使用 GCC 编译工具由 C 语言源程序生成可执行文件的过程并不单单是一个编译的过程，而要经过下面的几个过程。

- 预处理（Pre-Processing）
- 编译（Compiling）
- 汇编（Assembling）
- 链接（Linking）

在实际编译的时候，GCC 首先调用 cpp 命令进行预处理，主要实现对源代码编译前的预处理，比如将源代码中指定的头文件包含进来。接着调用 cc1 命令进行编译，作为整个编译过程的一个中间步骤，该过程会将源代码翻译生成汇编代码。汇编过程是针对汇编语言的步骤，调用 as 命令进行工作，生成扩展名为.o 的目标文件，当所有的目标文件都生成之后，

GCC 就调用链接器 ld 来完成最后的关键性工作——链接。

3.4.2 GCC 编译选项解析

GCC 是 Linux 下基于命令行的 c 语言编译器，其基本的使用语法如下。

gcc [option | filename]…

对于编译 C++的源程序，其基本的语法如下。

g++ [option | filename]…

其中 option 为 GCC 使用时的选项（后面会再详述），而 filename 为需要用 GCC 作编译处理的文件名。就 GCC 来说，其本身是一个十分复杂的命令，合理地使用其命令选项可以有效提高程序的编译效率、优化代码，GCC 拥有众多的命令选项，有超过 100 个的编译选项可用，按其应用有如下的分类。

1．常用编译选项

- -c 选项：这是 GCC 命令的常用选项。-c 选项告诉 GCC 仅把源程序编译为目标代码而并不做链接的工作，所以采用该选项的编译指令不会生成最终的可执行程序，而是生成一个与源程序文件名相同的以.o 为后缀的目标文件。例如一个 Test1.c 的源程序经过下面的编译之后会生成一个 Test1.o 的文件。

```
# gcc  –c  Test1.c
```

- -S 选项：使用该选项会生成一个后缀名为.s 的汇编语言文件，但是同样不会生成可执行的程序。
- -e 选项：-e 选项只对文件进行预处理，预处理的输出结果被送到标准输出（比如显示器）。
- -v 选项：在 Shell 的提示符号下键入 gcc -v，屏幕上就会显示出目前正在使用的 GCC 的版本信息。例如：

```
# gcc  -v
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux7/2.96/specs
gcc version 2.96 20000731（Redhat linux 7.3 2.96-128）
```

上面的系统信息指出了 GCC 的版本：gcc 2.96。

- -x language：强制编译器用指定的语言编译器来编译某个源程序。

例如下面的指令：

```
# gcc  –x  c++  P1.c
```

该指令表示强制采用 C++编译器来编译 C 程序 P1.c。

- -I<DIR>选项：库依赖选项，指定库及头文件路径。

在 Linux 下开发程序的时候，通常来讲都需要借助一个或多个函数库的支持才能够完成相应的功能。一般情况下，Linux 下的大多数函数都将头文件放到系统/usr/include/目录下，而库文件则放到/usr/lib/目录下。但在有些情况下并不是这样的，在这些情况下，使用 GCC

编译时必须指定所需要的头文件和库文件所在的路径。-I 选项可以向 GCC 的头文件搜索路径中添加新的目录<DIR>。例如，一个源程序所依赖的头文件在用户/home/include/目录下，此时就应该使用-I 选项来指定。

```
# gcc -I /home/include -o  Test  Test.c
```

- -L<DIR>：类似上面的情况，用来特别指定所依赖库所在的路径。

如果使用了不在标准位置的库，那么可以通过-L 选项向 GCC 的库文件搜索路径中添加新的目录。例如，一个程序要用到的库 libapp.so 在/home/zxq/lib/目录下，为了能让 GCC 能够顺利地链接该库，可以使用下面的命令：

```
#gcc –Test.c –L /home/zxq/lib –lapp –o Test
```

这里的-L 选项表示 GCC 去连接库文件 libapp.so。Linux 下的库文件在命名时有一个约定，那就是应该以 lib 三个字母开头，由于所有的库文件都遵循了同样的规范，因此在用-L 选项指定链接的库文件名时可以省去 lib 三个字母，也就是说 GCC 在对-lapp 进行处理时，会自动去链接名为 libapp.so 的文件。

- -static 选项：GCC 在默认情况下链接的是动态库，有时为了把一些函数静态编译到程序中，而无需链接动态库就采用-static 选项，它会强制程序链接静态库。
- -o 选项：在默认的状态下，如果 GCC 指令没有指定编译选项的情况下会在当前目录下生成一个名位 a.out 的可执行程序，例如：执行# gcc Test.c 命令之后会生成一个 a.out 的可执行程序。因此，为了指定生成的可执行程序的文件名，就可以采用-o 选项，比如下面的指令：

```
# gcc –o  Test  Test.c
```

执行该指令会在当前目录下生成一个名为 Test 的可执行文件。

注意 使用-o 选项时，-o 后面必须带有可执行文件的文件名（可以任意指定）。

2. 出错检查和警告提示选项

GCC 编译器包含完整的出错检查和警告提示功能，比如 GCC 提供了 30 多条警告信息和 3 个警告级别，使用这些选项有助于增强程序的稳定性和更加完善程序代码的设计，此类选项常用的如下。

- -pedantic　以 ANSI/ISO C 标准列出的所有警告

当 GCC 在编译不符合 ANSI/ISO C 语言标准的源代码时，如果在编译指令中加上了-pedantic 选项，那么源程序中使用了扩展语法的地方将产生相应的警告信息。

- -w　禁止输出警告消息
- -Werror　将所有警告转换为错误

Werror 选项要求 GCC 将所有的警告当成错误进行处理，这在使用自动编译工具（如 Make 等）时非常有用。如果编译时带上-Werror 选项，那么 GCC 会在所有产生警告的地方停止编译，只有程序员对源代码进行修改并且相应的警告信息消除时，才可能继续完成后续的编译

工作。

- -Wall　显示所有的警告消息

-Wall 选项可以打开所有类型的语法警告，以便于确定程序源代码是否是正确的，并且尽可能实现可移植性。

对 Linux 程序开发人员来讲，GCC 给出的警告信息是很有价值的，它们不仅可以帮助程序员写出更加健壮的程序，而且还是跟踪和调试程序的有力工具。建议在用 GCC 编译源代码时始终带上-Wall 选项，养成良好的习惯。

3．代码优化选项

代码优化指的是编译器通过分析源代码找出其中尚未达到最优的部分，然后对其重新进行组合，进而改善代码的执行性能。GCC 通过提供编译选项-On 来控制优化代码的生成，对于大型程序来说，使用代码优化选项可以大幅度提高代码的运行速度。

- -O 选项：编译时使用选项-O 可以告诉 GCC 同时减小代码的长度和执行时间，其效果等价于-O1。
- -O2 选项：选项-O2 告诉 GCC 除了完成所有-O1 级别的优化之外，同时还要进行一些额外的调整工作，如处理器指令调度等。

4．调试分析选项

- -g 选项：生成调试信息，GNU 调试器可利用该信息。GCC 编译器使用该选项进行编译时，将调试信息加入到目标文件当中，这样 gdb 调试器就可以根据这些调试信息来跟踪程序的执行状态。
- -pg 选项：编译完成之后，额外产生一个性能分析所需的信息。

注意　需要注意的是，使用调试选项都会使最终生成的二进制文件的大小急剧增加，同时增加程序在执行时的开销，因此调试选项通常推荐仅在程序的开发和调试阶段中使用。

下面举一个简单的例子来说明 GCC 的编译过程。首先用 vi 编辑器来编辑一个简单的 c 程序 test.c，程序清单如下。

```
#include <stdio.h>
int main()
{
 printf("Hello,this is a test!\n");
 return 0;
}
```

根据前面讲到的内容，使用 gcc 命令来编译该程序。

```
[root@localhost]# gcc –o test test.c
[root@localhost]#./test
Hello,this is a test!
```

可以从上面的编译过程看到，编译一个这样的程序非常简单，一条指令即可完成，事实上，这一条指令掩盖了很多细节。我们可以从编译器的角度来看上述的编译过程，这对于更好理解 GCC 编译工作原理有很好的帮助。

GCC 编译器首先做的工作是预处理：调用-E 参数可以让 GCC 在预处理结束后停止编译过程。

```
# gcc –E test.c -o test.i
```

编译器在这一步调用 cpp 工具来对源程序进行预处理，此时会生成 test.i 文件，下面部分列出了 test.i 文件中的内容。

```
# 1 "test.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "test.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 28 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 314 "/usr/include/features.h" 3 4
# 1 "/usr/include/sys/cdefs.h" 1 3 4
# 315 "/usr/include/features.h" 2 3 4
# 337 "/usr/include/features.h" 3 4
# 1 "/usr/include/gnu/stubs.h" 1 3 4
# 338 "/usr/include/features.h" 2 3 4
# 29 "/usr/include/stdio.h" 2 3 4

# 1 "/usr/lib/gcc/i386-redhat-linux/3.4.4/include/stddef.h" 1 3 4
# 213 "/usr/lib/gcc/i386-redhat-linux/3.4.4/include/stddef.h" 3 4
typedef unsigned int size_t;
# 35 "/usr/include/stdio.h" 2 3 4

# 1 "/usr/include/bits/types.h" 1 3 4
# 28 "/usr/include/bits/types.h" 3 4
# 1 "/usr/include/bits/wordsize.h" 1 3 4
# 29 "/usr/include/bits/types.h" 2 3 4
# 1 "/usr/lib/gcc/i386-redhat-linux/3.4.4/include/stddef.h" 1 3 4
# 32 "/usr/include/bits/types.h" 2 3 4
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
```

```
typedef unsigned long int __u_long;
typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
typedef unsigned short int __uint16_t;
typedef signed int __int32_t;
typedef unsigned int __uint32_t;
__extension__ typedef signed long long int __int64_t;
__extension__ typedef unsigned long long int __uint64_t;
```

查看代码会发现 stdio.h 的内容都被加入到该文件里去了，而且被预处理的宏定义也都作了相应的处理。

下一步是将 test.i 编译为目标代码，这可以通过使用-c 参数来完成。

```
#gcc -c test.i -o  test.o
```

GCC 默认将.i 文件看成是预处理后的 C 语言源代码，因此上述命令将自动跳过预处理步骤而开始执行编译过程，也可以使用-x 参数让 GCC 从指定的步骤开始编译。

编译的最后一步是将上一步所生成的目标文件链接成最终的可执行文件。

```
# gcc  test.o  -o  test
```

3.5 调试器 GDB 的使用技巧

3.5.1 GDB 调试器介绍

应用程序的调试是开发过程中必不可少的环节之一。Linux 下的 GNU 的调试器称为 GDB（GNU Debugger），该软件最早由 Richard Stallman 编写，GDB 是一个用来调试 C 和 C++ 程序的调试器（Debugger）。使用者能在程序运行时观察程序的内部结构和内存的使用情况，GDB 是一种基于命令行工作模式下的程序，工作在字符模式，由多个不同的图形用户界面前端予以支持，每个前端都能以多种方式提供调试控制功能，它的功能非常丰富，适用于修复程序代码中的问题，在 X Window 系统中，基于图形界面的调试工具称为 xxgdb。目前比较新的版本是 GDB6.4（2005 年 12 月 2 日发布），其官方网站是 http://www.gnu.org/software/gdb/。以下是 GDB 所提供的一些功能。

- 启动程序，并且可以设置运行环境和参数来运行指定程序。
- 让程序在指定断点处停止执行。
- 对程序做出相应的调整，这样就能纠正一个错误后继续调试。

需要注意的是，GDB 调试的是可执行文件，而不是源程序，如果想让 GDB 调试编译后生成的可执行文件，在使用 GDB 工具调试程序之前，必须使用带有-g 或-gdb 编译选项的 gcc 命令来编译源程序，例如：

```
# gcc -g -o test test.c
```

只有这样会在目标文件中产生相应的调试信息。调试信息包含源程序的每个变量的类型和在可执行文件里的地址映射以及源代码的行号，GDB 利用这些信息使源代码和机器码相关联。

使用 gdb 命令的语法如下。

```
# gdb  [参数]  Filename
```

下面列举一些常用的参数。

- -help：列出所有参数，并作简要说明。
- -symbols=file

-s file：读出文件（file）的所有符号。

- -core

-c

这里的 core 是程序非法执行后 core dump 后产生的文件。

- -directory

-d

加入一个源文件的搜索路径。默认搜索路径是环境变量中 PATH 所定义的路径。

- -quiet

  -q

使用该参数不显示 gdb 的介绍和版权信息等。

3.5.2 GDB 调试命令

运行 GDB 调试程序通常使用如下的命令。

```
# gdb  Filename
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu".
(gdb)
```

之后就可以在系统的（gdb）提示符后面输入相应的调试命令了，如果不希望出现 gdb 的系统信息提示，可以输入下面的指令：

```
# gdb   -q  Filename
```

表 3.7 列举了一些常用到的 GDB 调试命令。

表 3.7 常用 GDB 命令

命　令	说　明

file	指定要调试的可执行程序
kill	终止正在调试的可执行程序
next	执行一行源代码但并不进入函数内部
list	部分列出源代码
step	执行一行源代码并不进入函数内部
run	执行当前的可执行程序
quit	结束 gdb 调试任务
watch	可以检查一个变量的值而不管它何时被改变
print	打印表达式的值到标准输出
break N	在指定的第 N 行源代码设置断点
info break	显示当前断点清单，包括到达断点处的次数等
info files	显示被调试文件的详细信息
info func	显示所有的函数名
info local	显示当函数中的局部变量信息
info prog	显示被调试程序的执行状态
info var	显示所有的全局和静态变量名称
make	在不退出 gdb 的情况下运行 make 工具
shell	在不退出 gdb 的情况下运行 shell 命令
continue	继续执行正在调试的程序

下面举一个简单的例子来说明 GDB 调试命令的使用方法，下面的程序很简单，即通过用户输入一个圆的半径值来求得圆面积，其源代码如下。

```
#include<stdio.h>
#include<math.h>
int main(void)
{
  float Pi=3.1415926;
  float R;
  float S=0;
  printf("Please input your Ridus:\n");
  scanf("%f",&R);
  if (R>=0)
   {
     S=Pi*R*R;
     printf("The value of S is:%f\n",S);
   }
  else
```

```
    printf("Sorry,Wrong input!!\n");
  return 0;
}
```

为了方便调试可执行程序，可以用下面的语句来编译该程序。

```
# gcc -g -o new new.c
```

开始调试：

```
# gdb -q new
Using host libthread_db library "/lib/tls/libthread_db.so.1"
(gdb)
```

出现了（gdb）提示符以后，就可以输入相应的调试命令了。

1．查看源代码，使用 list 命令

```
(gdb) list
1   #include<stdio.h>
2   int main(void)
3   {
4   float Pi=3.1415926;
5   float R;
6   float S=0;
7   printf("Please input your Ridus:\n");
(gdb)
```

如上所示，使用 list 命令之后部分的列出了源代码，而且每行都有相应的标号，如果想列出更多的源代码，可以继续输入 list 命令（或者直接回车即可）。

2．运行该程序，使用 run 命令

```
(gdb) run
Starting program /home/zxq/new
Please input  your Radius:
10
The value of S is: 314.15926
Program exited normally
(gdb)
```

如上所示，使用 run 命令会执行编译后生成的可执行程序 new。

3．设置断点

gdb 可以使用 break N 命令来设置断点，N 表示在源代码的第 N 行处设置断点，例如：

```
(gdb) break 13
```

```
Breakpoint 1 at 0x804840a: file new.c,line 13.
```

这样程序执行到第 13 行语句处就会停止执行，

```
(gdb) run
Starting program: /home/zxq/new
Please input your Ridus:
9.5
Breakpoint 1, main () at new.c:13    /*指出程序执行停止的位置*/
13   printf("The value of S is:%f\n",S);
（gdb）
```

如果想看到程序中设置断点的数量或断点位置，可以使用 info break 命令来查看：

```
(gdb) info break
Num     Type      Disp Enb  Address  What
1   breakpoint       keep y  0x0804839c in main at new.c:4
2   breakpoint       keep y  0x08048426  in main at new.c:14
(gdb)
```

从上面的信息可以看到程序分别在第 4、14 行处设置了断点。

4．清除断点

c1ear 是一条用来清除断点的命令，在程序调试过程中，如果确定设置断点的语句处没有必要再暂停运行，就可以用 c1ear 命令来清除设置的断点。它的使用格式是：

```
（gdb） clear  n
```

在上述例子中，清除第 6 行处的断点的做法如下：

```
(gdb)  clear  13
Deleted breakpoint  l
```

事实上，比删除更好的一种方法是 disable 命令，关闭了断点，它并不会被删除，它只是让所设断点暂时失效，当还需要改断点时，enable 即可。

5．查看变量的值

当程序执行到断点处停止以后，往往要查看某些变量的值，进而观察程序的执行状态，gdb 采用 print 命令来查看指定变量的值，例如：

```
（gdb）break 13
Breakpoint 1 at 0x804840a: file new.c, line 13.
(gdb) run
Starting program: /home/zxq/new
Please input your Ridus:
9.5
Breakpoint 1, main () at new.c:13
```

```
13   printf("The value of S is:%f\n",S);
(gdb) print S          /*查看变量 S 的值*/
$1 = 283.528717
(gdb)
```

如果想看到变量的类型，可使用 whatis 命令，如：

```
(gdb) whatis S
type = float              /*变量 S 类型为 float*/
(gdb)
```

6．单步执行

gdb 提供以下两种方式。

step 指令，单步进入，可以跟踪到函数内部。命令是 step 或 s。

next 指令，单步，只是简单的单步执行，不会进入函数内部。

以上只是部分地列出了一些 GDB 调试指令的用法，事实上 GDB 具有非常的调试指令，具体详细的使用可参见 GNU GDB 使用手册。

7．搜索源代码

gdb 还提供了源代码搜索的命令。

向前搜索

```
(gdb) forward-search  <regexp>
(gdb) search <regexp>
```

全部搜索

```
(gdb) reverse-search <regexp>
```

其中，<regexp>就是正则表达式。

8．指定源文件的路径

某些时候，用-g 编译过后的执行程序中只是包括了源文件的名字，没有路径名。GDB 提供了可以指定源文件的路径的命令，以便 GDB 进行搜索要调试的源程序。

```
(gdb) dir <dirname ... >
```

9．结束当前程序的调试

kill 命令用来结束当前程序的调试。在 gdb 下直接输入下面这条命令即可结束程序的调试过程。

```
(gdb)  kill
Kill  programm being  debugged(y or n)
```

确认即可结束调试。

3.6 Linux 编程库

3.6.1 Linux 编程库介绍

所谓编程库就是指始终可以被多个 Linux 软件项目重复使用的代码集。以 C 语言为例，它包含了几百个可以重复使用的例程和调试程序的工具代码，其中包括函数。如果每次编写新程序都要重新写这些函数会非常不方便。使用编程库有两个主要优点。

- 可以简化编程，实现代码重复使用，进而减小应用程序的大小；
- 可以直接使用比较稳定的代码。

Linux 下的库文件分为共享库和静态库 2 大类，它们两者的差别仅在程序执行时所需的代码是在运行时动态加载的，还是在编译时静态加载的。此外，通常共享库以.so(Shared Object)结尾，静态链接库通常以.a 结尾（Archive）。在终端下查看库的内容，通常共享库为绿色，而静态库为黑色。

Linux 的库一般在/lib 或/usr/lib 目录下。它主要存放系统的链接库文件，没有该目录则系统无法正常运行。/lib 目录中存储着程序运行时使用的共享库。通过共享库，许多程序可以重复使用相同的代码，因此可以有效减小应用程序的大小。表 3.8 部分列出了一些 Linux 下常用到的编程库。

表 3.8　　常用到的 Linux 编程库

库　名　称	说　　明
libc.so	标准的 C 库
libdl.so	可以使用库的源代码而无需静态编译库
libglib.so	Glib 库
libm.so	标准数学库
libGL.so	OpenGL 的接口
libcom_err.so	常用出错例程集合
libdb.so	创建和操作数据库
libgthread.so	Glib 线程支持
libgtk.so	GIMP 下的 X 库
libz.so	压缩例程库
libvga.so	Linux 的 VGA 和 SVGA 图形库
libresolve.so	提供使用因特网域名服务器接口
libpthread.so	Linux 多线程库
libgdm.so	GNU 数据库管理器

3.6.2 Linux 系统调用

从字面意思上理解，系统调用说的是操作系统提供给用户程序调用的一组“特殊”接口。Linux 中用于创建进程的 fork()函数本身就是一个系统调用，使用系统主要目的是使得用户可以使用操作系统提供的有关设备管理、输入/输出系统、文件系统和进程控制、通信以及存储管理等方面的功能，而不必了解系统程序的内部结构和有关硬件细节，从而起到减轻用户负担和保护系统以及提高资源利用率的作用。

Linux 的运行空间划分为用户空间和内核空间，它们各自运行在不同的级别中，所以用户进程在通常情况下不允许访问内核，也无法使用内核函数，它们只能在用户空间操作用户数据，调用用户用空间函数。这样做的目的是为了对系统作必要的“保护”措施，但是使用系统调用可以最大程度地解决这一问题。其具体的措施是进程先用适当的值填充寄存器，然后调用一个特殊的指令，这个指令会跳到一个事先定义的内核中的一个位置（当然，这个位置是用户进程可读但是不可写的）。硬件知道一旦用户进程跳到这个位置，则认为该用户就不是在限制模式下运行的用户，而是作为操作系统的内核。当然，用户访问内核的路径是事先规定好的，只能从规定位置进入内核，而不允许任意跳入内核。

Linux 系统有 200 多个系统调用，这些系统调用按照功能分类大致可分为以下几个方面。

- 进程控制
- 文件系统控制
- 系统控制
- 内存管理
- 网络管理
- socket 控制
- 用户管理
- 进程间通信

类似于在 Windows 下面进行过 Win32 编程，windows 会提供 API（Application Programming Interface）接口函数作为 windows 操作系统提供给程序员的系统调用接口。同样的，Linux 作为一个操作系统也有它自己的系统调用，用户可以根据特定的方法来添加需要的系统调用。Linux 的 API 接口遵循 POSIX 标准，这套标准定义了一系列 API。在 Linux 中，这些 API 主要是通过 C 库（libc）实现的。下面通过举例来说明在 Linux 下添加新的系统调用的几个步骤。

（1）修改 kernel/sys.c，增加服务例程代码。首先编写添加到内核中的源程序，即要添加的服务，所用函数的名称应该是新的系统调用名称前面加上 sys_标志。例如新加的系统调用为 mysyscall（int number），那么就应该在系统的/usr/src/linux/kernel/sys.c 文件中添加相应的源代码，如下所示。

```
asmlinkage int sys_mysyscall(int number)
{
    printk（"This is a example of systemcall \n "）;
  return number;
}
```

为了说明问题，仅仅是一个返回一个值的简单例子。

注意 系统调用函数通常在成功时返回 0 值，不成功时返回非零值。

（2）添加新的系统调用后，为了从已有的内核程序中增加到新的函数的连接，需要编辑以下 2 个文件。

① /usr/src/linux/include/asm-i386/unistd.h

② /usr/src/linux/arch/i386/kernel/syscall_table.S

第 1 个文件中定义了每个系统调用的中断号，可以打开文件/usr/src/linux/include/asm-i386/unistd.h 来查看，该文件中包含了系统调用清单，用来给每个系统调用分配一个唯一的号码，部分内容如下。

```
............
#define __NR_add_key                 286
#define __NR_request_key             287
#define __NR_keyctl                  288
#define __NR_ioprio_set              289
#define __NR_ioprio_get              290
#define __NR_inotify_init            291
#define __NR_inotify_add_watch       292
#define __NR_inotify_rm_watch        293
#define  NR_syscalls                 294
............
```

文件中每一行的格式如下。

```
# define __NR_syscallname  N
```

syscallname 为系统调用名，而 N 则是该系统调用对应的中断号，每个系统调用都有唯一的中断号。应该将新的系统调用名称加到清单的最后，并给它分配号码序列中下一个可用的系统调用号。在该文件中的最后一句：#define NR_syscalls 294

NR_syscalls 表示系统调用的个数，294 表示有 294 个系统调用，标号从 0 开始，所以最后一个系统调用号是 293，那么如果新添加一个系统调用其中断号就应该是 294。

例如可以在该文件中这样定义一个系统调用：

```
# define    __NR __mysyscall   294
```

如果还需要添加另外的系统调用，可以此类推将中断号依次递增。此外需要注意的是，重新添加系统调用之后，应该在/usr/src/linux/include/asm-i386/unistd.h 文件中的#define NR_syscalls 语句重新指定的编号 n，例如在上面添加一个新的系统调用之后，该语句应该为：

```
# define       NR_syscalls     295
```

第 2 个要编辑的文件是：/usr/src/linux/arch/i386/kernel/syscall_table.S。该文件中定义了系统调用列表。在该文件中有以下类似的内容。

```
.data
ENTRY(sys_call_table)
    .long sys_restart_syscall  /* 0 - old "setup()" system call, used for restarting */
    .long sys_exit
    .long sys_fork
    .long sys_read
    .long sys_write
    .long sys_open      /* 5 */
    .long sys_close
    .long sys_waitpid
    .long sys_creat
    .long sys_link
    .long sys_unlink/* 10 */
............
```

在该文件中添加新的系统调用：

```
.long sys_mysyscall
```

（3）重新编译内核。添加好系统调用之后，需要重新编译内核，并且用新的内核来启动，此时，系统调用就添加好了，重新编译内核的过程在这里不做详细介绍。

（4）测试新的系统调用，编辑程序 test_call.c 如下。

```
#include <linux/unistd.h>
#include <stdio.h>
#include <errno.h>
_syscall1(int, mysyscall, int, num);  /*系统调用宏定义*/
int main(void)
{
          int n;
          n=mysyscall(10);     /*执行系统调用*/
          printf("n=%d\n",n);
          return 0;
}
```

编译并执行该程序。

```
# gcc –o  test_call  -I/usr/src/linux/include  test_call.c
```

```
# ./test_call
n=10
```

输出值正确，说明添加系统调用就成功了。

3.6.3 Linux 线程库

简单地讲，进程是资源管理的最小单位，线程是程序执行的最小单位。一个进程至少需要一个线程作为它的指令执行体，进程管理着资源（比如 cpu、内存、文件等），而将线程分配到某个 cpu 上执行。一个进程当然可以拥有多个线程。

Linux 是一个多用户多任务的操作系统。多用户是指多个用户可以在同一时间使用计算机系统；多任务是指 Linux 可以同时执行几个任务，它可以在还未执行完一个任务时又执行另一项任务。在操作系统设计上，从进程演化出线程，最主要的目的就是更好地支持多处理器以及减小（进程/线程）上下文切换开销。

现在，多线程技术已经被许多操作系统所支持，包括 Windows/Linux。现在有 3 种不同标准的线程库：WIN32，OS/2 和 POSIX。其中前两种只能用在它们各自的平台上（WIN32 线程仅能运行于 Windows 平台上，OS/2 线程运行于 OS/2 平台上）。POSIX（Portable Operating System Interface Standard，可移植操作系统接口标准）规范则是适用于各种平台，而且已经或正在所有主要的 Unix/Linux 系统上实现。

Linux 系统下的多线程遵循 POSIX 接口，称为 pthread。POSIX 标准由 IEEE 制定，并由国际标准化组织接受为国际标准。在 Linux2.6 内核版本之前，LinuxThreads 是现有 Linux 平台上使用最为广泛的线程库，它由 Xavier Leroy 负责开发完成，并已绑定在 Glibc 中发行。LinuxThreads 是一种面向 Linux 的 POSIX 1003.1c-pthread 标准接口。它所实现的就是基于核心轻量级进程的“一对一”线程模型，一个线程实体对应一个核心轻量级进程，而线程之间的管理在核外函数库中实现。使用 LinuxThreads 线程库创建和管理线程常用到下面几个函数。

- pthread_create()　创建新的线程

pthread_create()函数类似 fork()函数，完整的函数形式如下。

```
int pthread_create（pthread_t thread，const pthread_attr_t*attr，void*（*func）（void*），void *arg）
```

第 1 个参数是一个 pthread_t 型的指针用于保存线程 ID，以后对该线程的操作都要用 ID 来标示。每个 LinuxThreads 线程都同时具有线程 ID 和进程 ID，其中进程 ID 就是内核所维护的进程号，而线程 ID 则由 LinuxThreads 分配和维护。

第 2 个参数是一个 pthread_attr_t 的指针用于说明要创建的线程的属性，使用 NULL 表示要使用缺省的属性。

第 3 个参数指明了线程运行函数的起始地址，是一个只有一个（void *）参数的函数。

第 4 个参数指明了运行函数的参数，参数 arg 一般指向一个结构。

函数返回值类型为整数，当创建线程成功时，函数返回 0，若不为 0 则说明创建线程失败。创建线程成功后，新创建的线程则运行参数 3 和参数 4 确定的函数，原来的线程则继续运行下一行代码。

- pthread_join() 等待线程结束

pthread_join()函数用来挂起当前线程直到由参数 thread 指定的线程终止为止，完整的函数形式如下。

```
int pthread_join（pthread_t thread，void* *status ）
```

第 1 个参数为被等待的线程标识符，第 2 个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值。

函数返回值类型为整数，成功返回 0，错误返回非零值。

- pthread_self() 获取线程 ID

函数原型

```
pthread_t pthread_self(void)
```

函数返回本线程的 ID。

- pthread_detach() 用于让线程脱离

pthread_detach()函数用于将处于连接状态的线程变为脱离状态，函数完整的形式如下。

```
int pthread_detach（pthread_t thread）
```

函数返回值类型为整数，如果成功将线程转换为脱离态时返回 0，否则返回非零值。

- pthread_exit() 终止线程

pthread_exit()函数用来终止线程，函数完整形式如下。

```
pthread_exit（void *status）
```

参数 status 是指向线程返回值的指针。

下面通过一个简单的例子来介绍基于 POSIX 线程接口的 Linux 多线程编程，编写 Linux 下的多线程程序，需要使用头文件 pthread.h，连接时需要使用库 libpthread.a。以下是一个简单的例子。

```
/*mypthread.c*/
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
void *thread_function(void *arg)
{
  int i;
  for ( i=0; i<20; i++)
 {
     printf("This is a thread!\n");
 }
  return NULL;
}
```

```
int main(void)
 {
   pthread_t mythread;
   if ( pthread_create( &mythread, NULL, thread_function, NULL) )
{
    printf("error creating thread.");
    abort();
  }

  printf("This is main process!\n");
  if ( pthread_join ( mythread, NULL ) )
 {
   printf("error joining thread.");
   abort();
  }
  exit(0);
}
```

编译并执行该程序：

```
#gcc –lpthread –o mypthread mypthread.c
./mypthread
This is main process!
This is a thread!
```

一个线程实际上就是一个函数，创建后，改线程立即被执行。在上面的例程中，系统创建了一个主线程，又用 pthread_create 创建了一个新的子线程。

事实上，在 Linux 2.6 内核以前，Linux 把进程当作其调度实体，内核并不真正支持线程（轻量线程实现）。它提供了一个 clone()系统调用来创建一个调用进程的拷贝，这个拷贝与调用者共享地址空间。LinuxThreads 项目就是利用这个系统调用，完全在用户级模拟了线程。Linuxthread 线程库目前存在一些不足之处，比如在信号处理，任务调度，以及进程间同步原语等方面。在 Linux2.6.x 内核中，Linux 内核的调度性能得到了很大改进。Linux 重写了其线程库，使用 NPTL（Native Posix Thread Library）来取代受争议的 LinuxThreads 线程库，成为 glibc 的首选线程库，与此同时，最新发布的 glibc 2.4 版本正式采用 NTPL 作为 pthread 实现。

“黑色经典”系列之《嵌入式 Linux 系统开发技术详解——基于 ARM》

FARSIGHT

第 4 章　交叉开发环境

本章目标

本章内容包括嵌入式交叉开发环境的概念和配置，以及应用程序交叉开发和调试的方法。交叉开发环境是嵌入式 Linux 开发的基础，后续的开发过程几乎都是基于交叉开发环境的。因此，理解和掌握本章内容会大大方便嵌入式 Linux 开发。

- 交叉开发环境介绍
- 建立交叉开发环境
- 交叉调试应用程序

4.1 交叉开发环境介绍

本节将介绍交叉开发模型以及相关概念，为后面具体配置交叉开发环境做好概念上的准备。

4.1.1 交叉开发概念模型

嵌入式系统是专用计算机系统，它对系统的功能、可靠性、成本、体积、功耗等某些方面有严格的要求。例如：PDA 需要通过电池供电，需要尽可能降低功耗；网络交换机，不需要键盘显示等外围设备；还有大部分嵌入式设备没有磁盘等大容量存储设备。

电信服务器也属于嵌入式系统范畴，尽管配置了显示器、键盘、鼠标等计算机外设，但是它更注重系统的可靠性，而不是用户界面的可操作性。

由于嵌入式系统硬件上的特殊性，一般不能安装发行版的 Linux 系统。例如 Flash 存储空间很小，没有足够的空间安装；或者处理器很特殊，也没有发行版的 Linux 系统可用。所以需要专门为特定的目标板定制 Linux 操作系统，这必然需要相应的开发环境。于是人们想到了交叉开发模式。交叉开发模型如图 4.1 所示。

图 4.1 所示中，TARGET 就是目标板，HOST 是开发主机。在开发主机上，可以安装开发工具，编辑、编译目标板的 Linux 引导程序、内核和文件系统，然后在目标板上运行。通常这

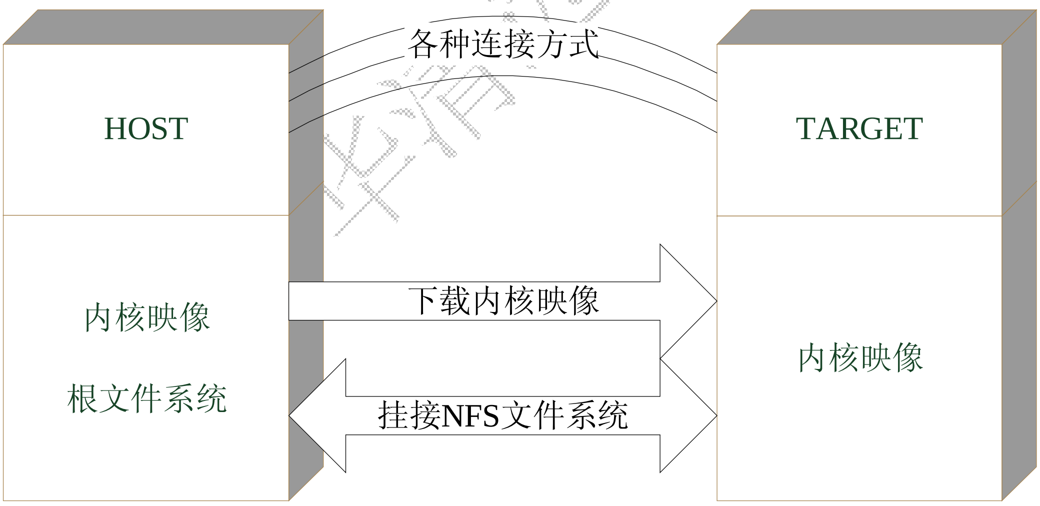


图 4.1　交叉开发模型

种在主机环境下开发，在目标板上运行的开发模式叫作交叉开发。

在交叉开发环境下，开发主机也是工作站，可以给开发者提供开发工具；同时也是一台服务器，可以配置启动各种网络服务。

在 PC 主机上，Linux 已经成为优秀的计算机操作系统。各种 Linux 发行版本，可以直接在 PC 上安装，功能十分强大。它不仅能够支持各种处理器和外围设备接口，而且提供了图形化的用户交互界面和丰富的开发环境，更重要的是 Linux 系统性能稳定。它为开发者提供

了以下功能。

- 非常稳定的多任务操作系统
- 丰富的设备驱动程序支持和网络工具
- 强大的 Shell
- 本地编译器
- 编辑器
- 图形化的用户界面

Redhat Linux 9 版本对计算机要求的最低配置如下。

- CPU 主频 400MHz 以上
- 内存 128MB
- 硬盘 1.3GB

推荐配置如下。

- CPU 主频 1.0GHz 以上
- 内存 256MB
- 硬盘 5GB

采用目前主流的计算机配置，完全能够满足推荐配置。无论 Linux 图形界面响应，还是程序编译，速度都很快，操作起来就很流畅。这对于嵌入式 Linux 开发者来说，可以大大提高开发效率。

对于交叉开发方式，一方面开发者可以在熟悉的主机环境下进行程序开发；另一方面又可以真实地在目标板系统上运行调试程序，可以避免受到目标板硬件的限制。这种开发方式贯穿嵌入式 Linux 系统开发的全过程。

要建立交叉开发方式，需要主机与目标板之间建立连接，才能实现远程通讯、传输文件等功能。这依赖于不同连接方式。

4.1.2 目标板与主机之间的连接

目标板和主机之间通常可以使用串口、以太网接口、USB 接口以及 JTAG 接口等连接方式。下面分别介绍这些通讯接口的特点。

（1）串行通讯接口

串行通讯接口常用 9 针串口（DB9）和 25 针串口（DB25），通信距离较近时（<12m），可以用电缆线直接连接标准 RS232C 端口；如果距离较远，就采用 RS422 或者 RS485 接口，需附加调制解调器（Modem）。其中最常用的是三线制接法，即地、接收数据和发送数据三脚相连，直接用 RS232C 相连，PC 机上一般带有 2 个 9 针串口。串口常用信号引脚如图 4.1 所示。

表 4.1　　串口常用信号引脚

引 脚 功 能	缩　　写	DB9 引脚号	DB25 引脚号
数据载波检测	DCD	1	8
接收数据	RXD	2	3
发送数据	TXD	3	2

数据终端准备	DTR	4	20
信号地	GND	5	7
数据设备准备好	DSR	6	6
请求发送	RTS	7	4
清除发送	CTS	8	5
振铃指示	DELL	9	22

通过串口可以作为控制台，向目标板发送命令，显示信息；也可以通过串口传送文件；还可以通过串口调试内核及程序。串口的设备驱动实现也比较简单。

缺点是通讯速率慢，不适合大数据量传输。

（2）以太网接口

以太网以其高度灵活，相对简单，易于实现的特点，成为当今最重要的一种局域网建网技术。虽然其他网络技术也曾经被认为可以取代以太网的地位，但是绝大多数的网络管理人员仍然把以太网作为首选的网络解决方案。

以太网 IEEE 802.3 通常使用专门的网络接口卡或通过系统主电路板上的电路实现。以太网使用收发器与网络媒体进行连接。收发器可以完成多种物理层功能，其中包括对网络碰撞进行检测。收发器可以作为独立的设备通过电缆与终端站连接，也可以直接被集成到终端站的网卡当中。

以太网采用广播机制，所有与网络连接的工作站都可以看到网络上传递的数据。通过查看包含在帧中的目标地址，确定是否进行接收或放弃。如果证明数据确实是发给自己的，工作站将会接收数据并传递给高层协议进行处理。

以太网采用 CSMA/CD 媒体访问机制，任何工作站都可以在任何时间访问网络。在发送数据之前，工作站首先需要侦听网络是否空闲，如果网络上没有任何数据传送，工作站就会把所要发送的信息投放到网络当中。否则，工作站只能等待网络下一次出现空闲的时候再进行数据的发送。

作为一种基于竞争机制的网络环境，以太网允许任何一台网络设备在网络空闲时发送信息。因为没有任何集中式的管理措施，所以非常有可能出现多台工作站同时检测到网络处于空闲状态，进而同时向网络发送数据的情况。这时，发出的信息会相互碰撞而导致损坏。工作站必须等待一段时间之后，重新发送数据。补偿算法用来决定发生碰撞后，工作站应当在何时重新发送数据帧。

网络接口一般采用 RJ-45 标准插头，PC 机上一般都配置 10M/100M 以太网卡，实现局域网连接。通过以太网连接和网络协议，可以实现快速的数据通讯和文件传输。

缺点是驱动程序实现比较麻烦，好在以太网接口的设备驱动也很多。

（3）USB 接口

USB（Universal Serial Bus）接口支持热插拔，具有即插即用的优点，最多可连接 127 台外设，所以 USB 接口已经成为 PC 外设的标准接口。USB2 有两个规范，即 USB 1.1 和 USB 2.0。

USB 1.1 是较早的 USB 规范，其高速方式的传输速率为 12Mbps，低速方式的传输速率为 1.5Mbps。

USB 2.0 规范则是最新的 USB 规范。它的传输速率达到了 480Mbps，足以满足大多数外设的速率要求。USB 2.0 中的“增强主机控制器接口”（EHCI）定义了一个与 USB 1.1 相兼容的架构。所有支持 USB 1.1 的设备都可以直接在 USB 2.0 的接口上使用而不必担心兼容性问题，而且像 USB 线、插头等附件也都可以直接使用。

USB 的设备支持热插拔，通讯速率也很快。

缺点是 USB 设备区分主从端，两端分别要有不同的驱动程序支持。

（4）JTAG 等接口

JTAG 技术是一种嵌入式调试技术，它在芯片内部封装了专门的测试电路测试接口（TAP，Test Access Port），通过 JTAG 测试工具对芯片的核进行测试。它是联合测试行动小组（JTAG，Joint Test Action Group）定义的一种国际标准测试协议，主要用于芯片内部测试及对系统进行仿真、调试。

目前大多数比较复杂的器件都支持 JTAG 协议，如 ARM、DSP、FPGA 器件等。标准的 JTAG 接口是 4 线：TMS、TCK、TDI、TDO，分别为测试模式选择、测试时钟、测试数据输入和测试数据输出。

JTAG 接口的时钟一般在 1MHz～16MHz 之间，所以传输速率可以很快。但是实际的数据传输速度要取决于仿真器与主机端的通讯速度和传输软件。

另外还有 EJTAG（Extended JTAG）和 BDM（Background Debug Mode）接口定义，分别在 MIPS 芯片和 PowerPC 5xx/8xx 芯片上设计应用。这些接口的电气性能不同，但是功能大体上是相似的。

4.1.3 文件传输

主机端编译的 Linux 内核影像必须有至少一种方式下载到目标板上执行。通常是目标板的引导程序负责把主机端的影像文件下载到内存中。根据不同的连接方式，可以有多种文件传输方式，每一种方式都需要相应的传输软件和协议。

（1）串口传输方式

主机端通过 kermit、minicom 或者 windows 超级终端等工具都可以通过串口发送文件。当然发送之前需要配置好数据传输率和传输协议，目标板端也要做好接收准备。通常波特率可以配置成 115200bit/s，8 位数据位，不带校验位。传输协议可以是 Kermit、Xmodem、Ymodem、Zmodem 等。

（2）网络传输方式

网络传输方式一般采用 TFTP（Trivial File Transport Protocol）协议。TFTP 协议是一种简单的网络传输协议，是基于 UDP 传输的，没有传输控制，所以对于大文件的传输是不可靠的。不过正好适合目标板的引导程序，因为协议简单，功能容易实现。当然，使用 TFTP 传输之前，需要驱动目标板以太网接口并且配置 IP 地址。

（3）USB 接口传输方式

通常分主从设备端，主机端为主设备端，目标板端为从设备端。主机端需要安装驱动程序，识别从设备后，可以传输数据。USB 2.0 标准的数据传输速率非常快。

（4）JTAG 接口传输方式

JTAG 仿真器跟主机之间的连接通常是串口、并口、以太网接口或者 USB 接口。传输速率也受到主机连接方式的限制，这取决于仿真器硬件的接口配置。

采用并口连接方式的仿真器最简单，也叫作 JTAG 电缆（CABLE），价格也最便宜。性能好的仿真器一般会采用以太网接口或者 USB 接口通信。

（5）移动存储设备

如果目标板上有软盘、CDROM、USB 盘等移动存储介质，就可以制作启动盘或者复制到目标板上，从而引导启动。移动存储设备一般在 X86 平台上比较普遍。

4.1.4 网络文件系统

网络文件系统（NFS，Network File System）最早是 SUN 开发的一种文件系统。NFS 允许一个系统在网络上共享目录和文件。通过使用 NFS，用户和程序可以像访问本地文件一样访问远端系统上的文件，这极大地简化了信息共享。

Linux 系统支持 NFS，并且可以配置启动 NFS 网络服务。

NFS 文件系统的优点如下。

（1）本地工作站使用更少的磁盘空间，因为通常的数据可以存放在一台机器上而且可以通过网络访问到。

（2）用户可以通过网络访问共享目录，而不必在计算机上为每个用户都创建工作目录。

（3）软驱、CDROM 等存储设备可以在网络上面共享使用。这可以减少整个网络上的移动介质设备的数量。

（4）NFS 至少有一台服务器和一台（或者更多）客户机两个主要部分。客户机远程访问存放在服务器上的数据。需要配置启动 NFS 等相关服务。

网络文件系统的优点正好适合嵌入式 Linux 系统开发。目标板没有足够的存储空间，Linux 内核挂接网络根文件系统可以避免使用本地存储介质，快速建立 Linux 系统。这样可以方便地运行和调试应用程序。

4.2 安装交叉编译工具

基于上述硬件环境配置的需求，接下来一步步构建这个交叉开发环境。首先要安装交叉编译工具链。

4.2.1 获取交叉开发工具链

Linux 使用 GNU 的工具，社区的开发者已经编译出了常用体系结构的工具链，从因特网上可以下载。我们可以下载这些工具，建立交叉开发环境，也可以自己动手编译新的工具链，那就请仔细阅读第 5 章的内容。Linux 公司会更加重视工具链的维护。

对于 ARM 体系结构的编译器，也有不少站点提供下载。免费提供的工具链包括 binutils

和 gcc，但是都不提供 gdb 交叉调试器。社区主要维护 Linux 内核的发布，文件系统也很少。这里介绍几个 ARM Linux 的免费站点。

（1）http://arm.linux.org.uk

这个站点是 ARM Linux 的官方站点，Linux 2.4 内核发布过很多针对 ARM 平台的补丁。有许多 ARM/XSCALE 开发者维护这个站点，也可以下载到 ARM/XSCALE 开发常用的工具链。

ARM Linux 工具链下载的 HTTP 和 FTP 地址如下。

http://ftp.arm.linux.org.uk/pub/armlinux/toolchain/

ftp://ftp.arm.linux.org.uk/pub/linux/arm/toolchain/

（2）http://www.handhelds.org

HANDHELDS 是手持设备的开发网站。因为 ARM/XSCALE 处理器在手持设备上应用广泛，所以也有很多 ARM Linux 开发的资源。

这里的 ARM Linux 工具链版本比其他网站相对高一些，下载链接地址如下。

http://www.handhelds.org/download/projects/toolchain/

（3）http://linux.omap.com

这是 OMAP Linux 网站，从 TI 公司网站可以链接过来。TI 公司基于 ARM926E 核发布了一系列 OMAP 处理器，具有低功耗，智能电源管理的特点，适合移动手持设备的应用。这个站点专门为 OMAP 平台提供 Linux BSP。

这里的 ARM 工具链下载连接地址如下。

http://linux.omap.com/pub/toolchain

（4）http://www.mvista.com

这是 Montavista 公司的主页网址，浏览 Professional 3.1 的版本的产品介绍，会发现可以免费注册部分平台的预览版（Previewkit）。注册成功以后，通过 Email 得到 Montavista 提供的下载网址和软件安装密码。

Montavista Linux 能够支持各种体系结构的开发板，只对部分硬件平台提供预览版。预览版是 Montavista 免费提供给客户的软件，包括交叉编译器、内核以及很小的一个文件系统，可以用来学习建立嵌入式 Linux 交叉开发环境。

Montavista Linux 的发行版包含完整的交叉开发工具链、内核和文件系统，还有集成开发环境。Montavista Linux 发行版对内核、应用程序和库、以及工具都已经作过完整的测试，对产品提供技术支持等。

另外，还有其他 Linux 发行商的开发包。有些半导体商也会为他们的处理器提供板级开发包（BSP，Board Support Packages）。

4.2.2 主机安装工具链

下载的工具链有不同的包装格式，RPM 的格式就很常用，也有把工具链直接压缩成 tar 包的。

对于 RPM 的格式，可以通过 rpm 命令把软件包安装到主机上。可是这些工具安装到哪里去了呢？RPM 包安装的时候都会有缺省的安装目录，可以通过 rpm 命令来查询。这个命令

是 Redhat Linux 上的常用命令，可以参考第 3 章的内容。

对于 tar 包，可以使用 tar 命令解压的。问题是解压出来的工具应该放在什么路径下？因为 GCC 编译器的运行是依赖于其他工具和库，通常不能把这些工具放在任意目录下。只好向下载的站点求教，一般通过相关的 README 或者说明文档可以得到具体的安装路径。

另外，通过 gcc 命令也可以得到安装的路径。以 ARM Linux 站点提供的 cross-3.3.2.tar.bz2 包为例说明。解压 cross-3.3.2.tar.bz2 后，查看 GCC 版本号，可以得到一些信息。

```
$ tar –jxvf  cross-3.3.2.tar.bz2
$ ./3.3.2/bin/arm-linux-gcc –v
Reading specs from ./3.3.2/bin/../lib/gcc-lib/arm-linux/3.3.2/specs
Configured with: ../gcc-3.3.2/configure --target=arm-linux
--with-cpu=strongarm1100 --prefix=/usr/local/arm/3.3.2 i686-pc-linux-gnu
--with-headers=/work/kernel.h3900/include --enable-threads=pthreads
--enable-shared --enable-static --enable-languages=c,c++
Thread model: posix
gcc version 3.3.2
```

从上面打印的版本信息中可以看到“--prefix=/usr/local/arm/3.3.2”，这就是 GCC 安装的路径。它是在 GCC 编译前通过 prefix 选项配置的。

所以，这个工具链应该安装的路径是：/usr/local/arm/3.3.2。

```
$ mkdir –p /usr/local/arm
$ mv ./2.95.3 /usr/loca/arm/
```

然后，在环境变量 PATH 中添加路径，就可以直接使用 arm-linux-gcc 命令了。

```
$ export PATH=$PATH:/usr/local/arm/3.3.2/bin
```

4.3 主机开发环境配置

4.3.1 主机环境配置

主机端安装 Linux 操作系统的时候，只要磁盘有足够空间，最好是完全安装。因为漏装了有些软件工具，会使得开发很不方便。当然，安装完了以后再来安装需要的软件包也可以，最好是直接使用 rpm 命令来安装对应的包。

接下来就是主机 Linux 环境配置。首先要确认主机的网络接口驱动成功，并且配置网络接口的 IP 地址。可以通过 ifconfig 命令查看所有网络接口，还可以配置网口的 IP 地址。

```
$ ifconfig –a
```

```
eth0      Link encap:Ethernet  HWaddr 00:0E:A6:B4:56:E6
          inet addr: 192.168.254.1  Bcast: 192.168.254.255  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen: 0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
          Interrupt:0 Base address:0xa000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:241 errors:0 dropped:0 overruns:0 frame:0
          TX packets:241 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:15950 (15.5 Kb)  TX bytes:15950 (15.5 Kb)
$ ifconfig eth0 192.168.254.1
```

也可以通过 Redhat Linux 9 的图形配置界面来配置，启动配置窗口的命令为 redhat-config-network。图 4.2 所示就是网络设备配置的图形窗口。

然后把交叉开发工具链的路径添加到环境变量 PATH 中，这样可以方便地在 Bash 或者 Makefile 中使用这些工具。通常可以在环境变量的配置文件有 3 个，分别在不同的范围生效。

/etc/profile 是系统启动过程执行的一个脚本，对所有用户都生效。

～/.bash_profile 是用户的脚本，在用户登录时生效。

～/.bashrc 也是用户的脚本，在～/.bash_profile 中调用生效。

把环境变量配置的命令添加到其中一个文件中即可。

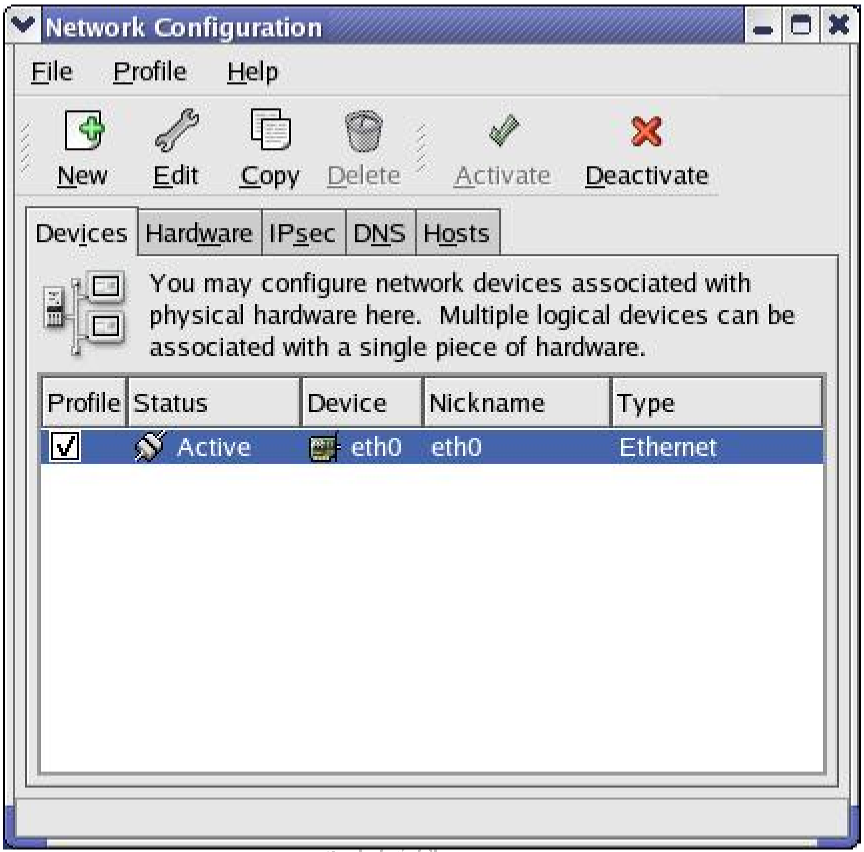


图 4.2 网络设备配置窗口

4.3.2 串口控制台工具

串行通讯接口很适合作为控制台，在各种操作系统上一般都有现成的控制台程序可以使用。Windows 操作系统有超级终端（Hyperterminal）工具；Linux/UNIX 操作系统有 minicom 等工具。

无论什么操作系统和通讯工具，都可以作为串口控制台。如果在 Windows 平台上运行 Linux 虚拟机，这个串口通讯软件可以任选一种。

超级终端是 Windows 系统的串口通讯工具，完全图形化的界面，操作非常简单。使用超级终端也要配置相应的连接。

建立一个超级终端的连接，需要为其配置如图 4.3 所示的参数。主要是串口号、通讯速



图 4.3　超级终端配置界面

率和是否流控。每建立一个配置可以保存下来。

Linux 系统通常使用 minicom 串口通讯工具。由于 minicom 不是图形窗口的工具，操作起来要麻烦一些。使用 minicom 串口终端之前，需要先配置参数。

Minicom 的配置界面是菜单方式。在 Shell 下执行“minicom –s”命令，出现如图 4.4 所示的配置菜单。注意 minicom 程序要访问串口设备，需要以 root 的权限操作。

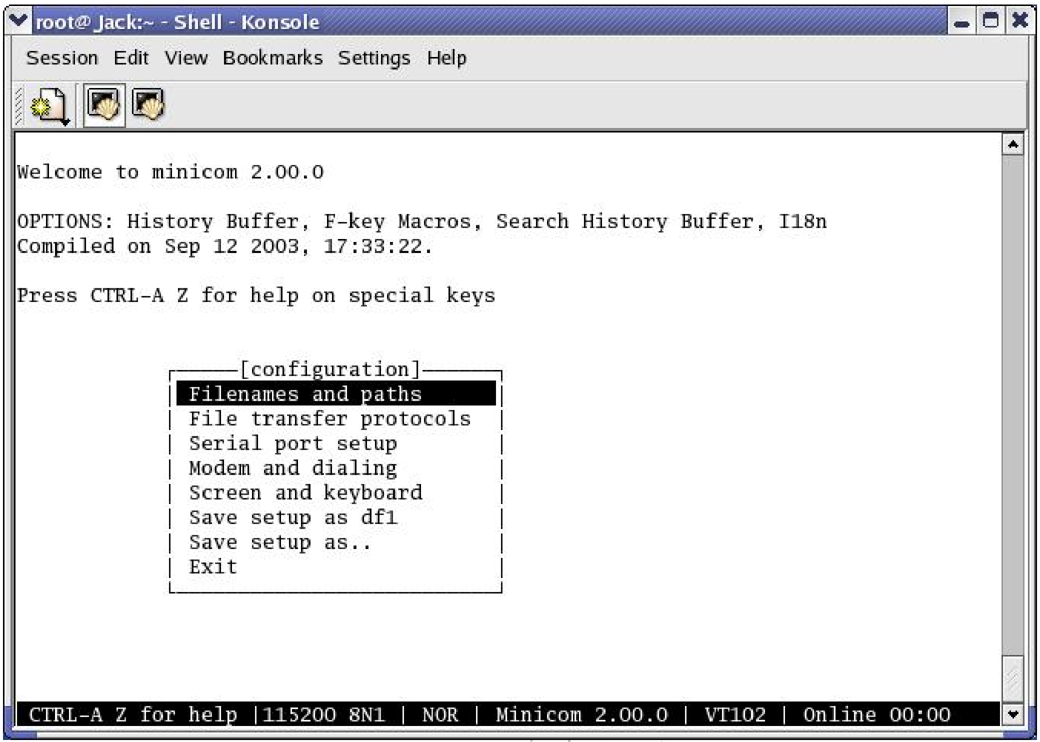


图 4.4 minicom 配置界面

图 4.4 菜单中，可以先通过光标移动键选中菜单项，再敲回车进入子菜单项。

选择“Serial port setup”菜单项，根据目标板的串口通讯参数设置。这些配置项都有快捷键（用大写字母显示），可以通过相应的按键选择进入子项。串口配置参数如图 4.5 所示。

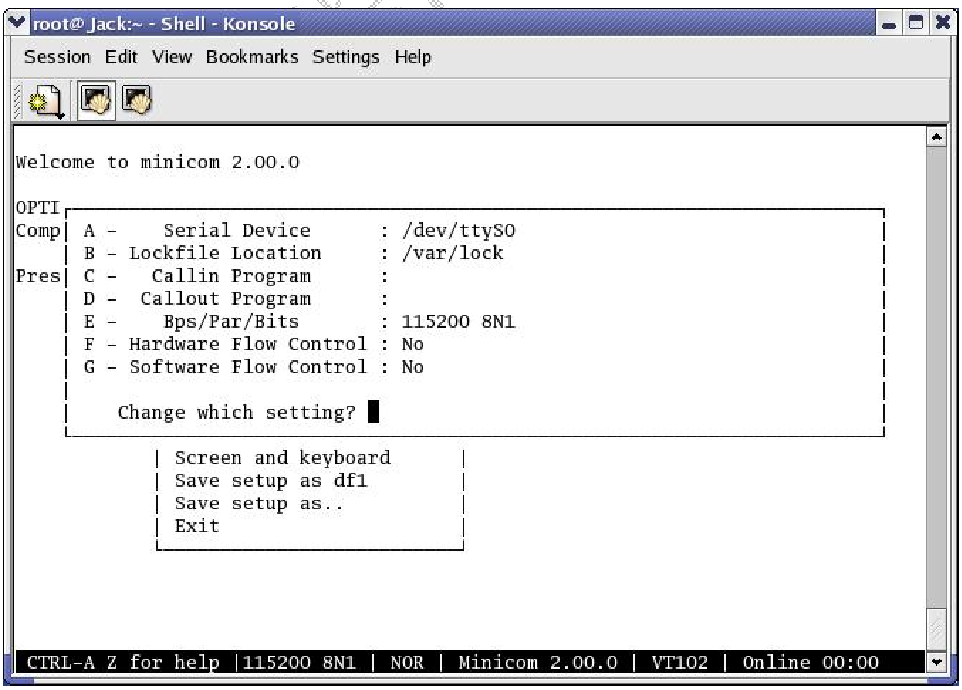


图 4.5　minicom 串口参数配置界面

敲【A】键，可以进入并且修改要使用的串口设备，例如：/dev/ttyS0 是串口 1，/dev/ttyS1 是串口 2。修改完一项，按【ESC】键返回准备选择其他配置项。通常串口通讯速率和硬件流控也要设置，这些项在配置时提供可选的参数值。

参数设置完成后，敲回车键返回如图 4.4 所示的主配置菜单。这时可以保存配置参数。移动光标选择“Save as dfl”菜单项，敲回车保存为缺省设置。

最后移动光标选择“Exit from Minicom”退出。

Minicom 的配置参数缺省保存在/etc/minirc.dfl 文件中，内容如下。

```
#/etc/minirc.dfl
# Machine-generated file - use "minicom -s" to change parameters.
pr port             /dev/ttyS0
pu baudrate         115200
pu minit
pu mreset
pu mhangup
pu rtscts           No
```

再启动 minicom 的时候，直接在 Shell 下执行 minicom 命令，就可以进入 minicom 控制台。

当运行在 minicom 控制台下面时，通过组合键可以进入 minicom 菜单。组合键的用法是：先按【Ctrl+A】组合键，再敲入一个命令键。其中主要的几个命令键如下。

【Z】命令键是显示所有的命令并进入命令主菜单，如图 4.6 所示。

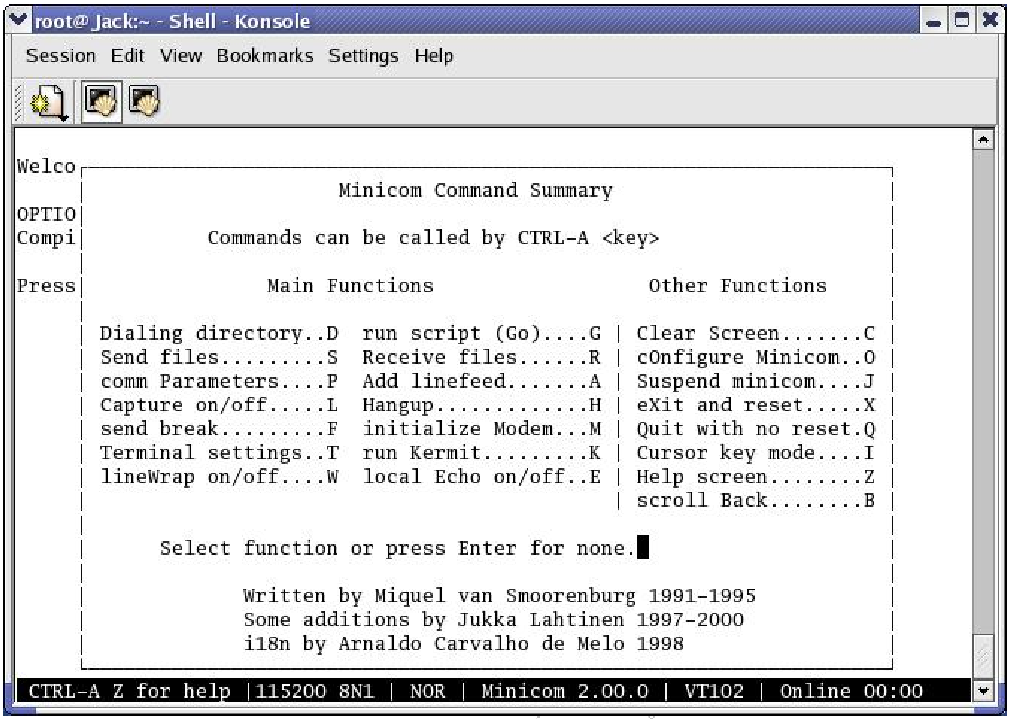


图 4.6 minicom 命令主菜单

【X】命令键退出 minicom，会提示确认退出；

【ESC】键退出命令主菜单，返回到控制台。

4.3.3 DHCP 服务

目标板的 Bootloader 或者内核都需要分配 IP 地址。这可以通过动态主机配置协议（DHCP Dynamic Host Configuration Protocol）或者 BOOTP 协议实现。

BOOTP 协议可以给计算机分配 IP 地址并且通过网络获取映像文件的路径；DHCP 则是向后兼容 BOOTP 的协议拓展。

Linux 操作系统的主机一般包含 dhcpd 的软件包，可以配置 DHCP 服务。配置服务的操作需要 root 用户的权限。

首先要确认主机上已经安装所有必需的软件包，创建相关文件。确认/var/lib/dhcp/dhcpd.leases 已经存在。如果这个文件不存在，可以手工创建目录和文件。

DHCP 服务的配置文件是/etc/dhcpd.conf，通过“man dhcpd.conf”命令可以查看配置手册。手册详细说明了 dhcpd.conf 几种配置语句的用法。以下一个很好的例子。

```
# /etc/dhcpd.conf
allow bootp;
ddns-update-style none;
subnet 192.168.1.0 netmask 255.255.255.0 {
     group {
          host mytarget {
```

```
            hardware ethernet 00:01:EC:E0:0A:0B;
            fixed-address 192.168.1.100;
            filename "zImage";
            option root-path "/usr/local/arm/3.3.2/rootfs";
            }
        }
}
```

上面的配置文件中，为指定的目标板配置了相关网络参数。其中有些参数含义如下。

（1）host 指定目标板网络名称为“mytarget”，在直接使用 IP 地址的局域网没有什么影响。mytarget 可以是目标板的网络名称。

（2）hardware ethernet 对应目标板以太网接口的 MAC 地址。这个需要在目标板网络接口打开以后获取相关参数，然后修改配置。

（3）fixed-address 是给目标板分配的 IP 地址。通常是指定的一个 IP 地址。

（4）filename 是映像文件的名称。目标板的 Bootloader 可以通过 BOOTP 协议获取映像文件，然后下载到目标板内存。这一项并不是所有目标板必要的。

（5）root-path 是网络文件系统的路径，目标板可以根据这个路径挂接 NFS 根文件系统。

（6）subnet 和 netmask 分别是子网和掩码，IP 地址的配置需要在这个网段。

配置好 dhcpd.conf 文件以后，就可以启动 dhcpd 守候进程了。可以通过图形化的服务配置界面。命令行的方式也很简洁，执行启动命令：

```
$ /etc/init.d/dhcpd start
```

每次修改 dhcpd.conf 文件以后，都需要重启 dhcpd 服务。执行下列命令。

```
$ /etc/init.d/dhcpd restart
```

如果希望系统每次重启都自动启动这项服务，可以使用 chkconfig 命令打开配置。

```
$ chkconfig dhcpd on
```

这样 DHCP 服务就设置完成了。在使用过程中，可能需要经常修改配置文件并且重启 dhcpd 服务。

网络服务的启动和停止也可以通过图形化窗口来配置，在 Redhat Linux 9 系统上可以执行“redhat-config-network”命令，弹出图 4.7 所示配置窗口。

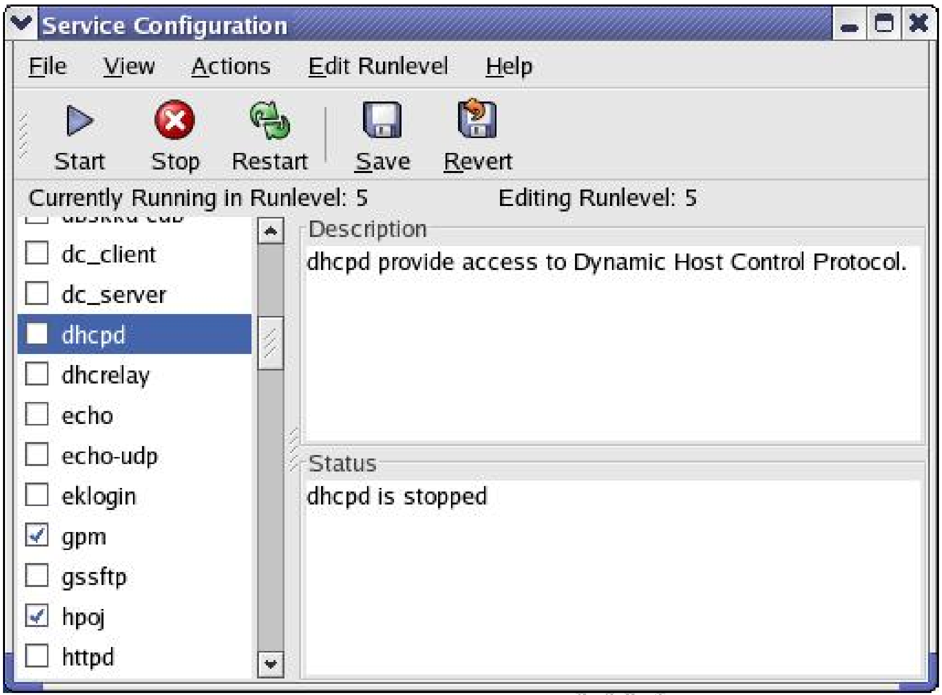


图 4.7 系统服务配置窗口

4.3.4 TFTP 服务

TFTP 协议是简单的文件传输协议，所以实现简单，使用方便，正好适合目标板 Bootloader 使用。但是文件传输是基于 UDP 的，文件传输（特别是大文件）是不可靠的。

TFTP 服务在 Linux 系统上有客户端和服务器 2 个软件包。配置 TFTP 服务，必须先安装好。

TFTP 服务也可以通过图形化的配置窗口来启动。当然操作过程需要 root 权限。缺省的情况下，把/tftpboot 目录作为输出文件的根目录。

另外，还可以手工修改 TFTP 配置文件，定制 TFTP 服务。通过命令行的方式启动 TFTP 服务。配置文件/etc/xintd.d/tftp 内容如下。

```
# /etc/xintd.d/tftp
# default: off
service tftp
{
     disable    =   yes
     socket_type        = dgram
     protocol       = udp
     wait           = yes
     user           = root
     server         = /usr/sbin/in.tftpd
     server_args        = -s /tftpboot
```

```
        per_source     = 11
        cps            = 100 2
        flags          = IPv4
    }
```

其中，disable 是指关闭还是打开 tftp 服务。如果要打开服务，把 yes 改成 no。server 是指定服务器程序为/usr/sbin/in.tftpd。server_args 则指定输出文件的根目录为/tftpboot，文件必须放到/tftpboot 目录下才能被输出。

修改配置以后，还需要执行下列命令使 xinetd 重新启动 TFTP 服务。

```
$ /etc/init.d/xinetd restart
```

4.3.5　NFS 服务

NFS 服务的主要任务是把本地的一个目录通过网络输出，其他计算机可以远程地挂接这个目录并且访问文件。

NFS 服务有自己的协议和端口号，但是在文件传输或者其他相关信息传递的时候，NFS 则使用远程过程调用（RPC，Remote Procedure Call）协议。

RPC 负责管理端口号的对应与服务相关的工作。NFS 本身的服务并没有提供文件传递的协议，它通过 RPC 的功能负责。因此，还需要系统启动 portmap 服务。

NFS 服务通过一系列工具来配置文件输出，配置文件是/etc/exports。配置文件的语法格式如下。

```
共享目录 主机名称 1 或 IP1(参数 1，参数 2) 主机名称 2 或 IP2(参数 3，参数 4)
```

“共享目录”是主机上要向外输出的一个目录；

“主机名称或者 IP”则是允许按照指定权限访问这个共享目录的远程主机；

“参数”则定义了各种访问权限。

exports 配置文件参数说明如表 4.2 所示。

表 4.2　　exports 配置文件参数说明

参　　数	含　　义
rw	具有可擦写的权限
ro	具有只读的权限
no_root_squash	如果登录共享目录的使用者是 root 的话，那么他对于这个目录具有 root 的权限
root_squash	如果登录共享目录的使用者是 root 的话，那么他的权限将被限制为匿名使用者，通常他的 UID 与 GID 都会变成 nobody
all_squash	不论登录共享目录的使用者是什么身份，他的权限将被限制为匿名使用者
anonuid	前面关于*_squash 提到的匿名使用者的 UID 设定值，通常为 nobody。这里可以设定 UID 值，并且 UID 也必须/etc/passwd 中设置

anongid	与上面的 anonuid 类似，只是 GID 变成 group ID
sync	文件同步写入到内存和硬盘当中
async	文件会先暂存在内存，而不是直接写入硬盘

举例说明如下。

（1）/usr/local/arm/3.3.2/rootfs *(rw, no_root_squash)

表示输出/usr/local/arm/3.3.2/rootfs 目录，并且所有的 IP 都可以访问。

（2）/home/public 192.168.0.*(rw)

表示输出/home/public 目录，只允许 192.168.0.*网段的 IP 访问。

（3）/home/test 192.168.1.100(rw)

表示输出/home/test 目录，并且只允许 192.168.1.100 访问。

（4）/home/linux *.linux.org（rw, all_squash, anonuid=40, anongid=40）

表示输出/home/linux 目录，并且允许*.linux.org 主机登录。在/home/linux 下面写文件时，文件的用户变成 UID 为 40 的使用者。

编辑修改好/etc/exports 这个配置文件，就可以启动服务 portmap 和 nfs 服务了。常用系统启动脚本来启动服务。

```
$ /etc/rc.d/init.d/portmap start
$ /etc/rc.d/init.d/nfs start
```

也可以通过 service 命令来启动。

```
$ service nfs start
$ service portmap start
```

启动完成后，可以查看/var/log/messages，确认是否正确激活服务。

如果只修改了/etc/exports 文件，并不总是要重启 NFS 服务。可以使用 exportfs 工具重新读取/etc/exports，就可以加载输出的目录。

exportfs 工具的使用语法如下。

```
exportfs [-aruv]
```

-a：全部挂载（或卸载）/etc/exports 的设置。

-r：重新挂载/etc/exports 的设置，更新/etc/exports 和/var/lib/nfs/xtab 里面的内容。

-u：卸载某一个目录。

-v：在输出的时候，把共享目录显示出来。

在 NFS 已经启动的情况下，如果又修改了/etc/exports 文件，可以执行命令：

```
$ exportfs –ra
```

系统日志文件/var/lib/nfs/xtab 中可以查看共享目录访问权限，不过只有已经被挂接的目录才会出现在日志文件中。

远程计算机作为 NFS 客户端，可以简单通过 mount 命令挂接这个目录使用。例如：

```
$ mount -t nfs 192.168.1.1:/home/test /mnt
```

这条命令就是把 192.168.1.1 主机上的/home/test 目录作为 NFS 文件系统挂接到/mnt 目录下。如果系统每次启动的时候都要挂接，可以在 fstab 中添加相应一行配置。

如果希望 NFS 服务在每次系统引导时都要启动，可以通过 chkconfig 打开这个选项。

```
$ /sbin/chkconfig nfs on
```

4.4 启动目标板

4.4.1 系统引导过程

在各种体系结构平台上，多数内核映像都采用压缩格式（MIPS 平台例外，它的映像采用非压缩格式）。Linux 系统的一般启动过程通常划分为内核引导、内核启动和应用程序启动 3 个阶段，如图 4.8 所示。

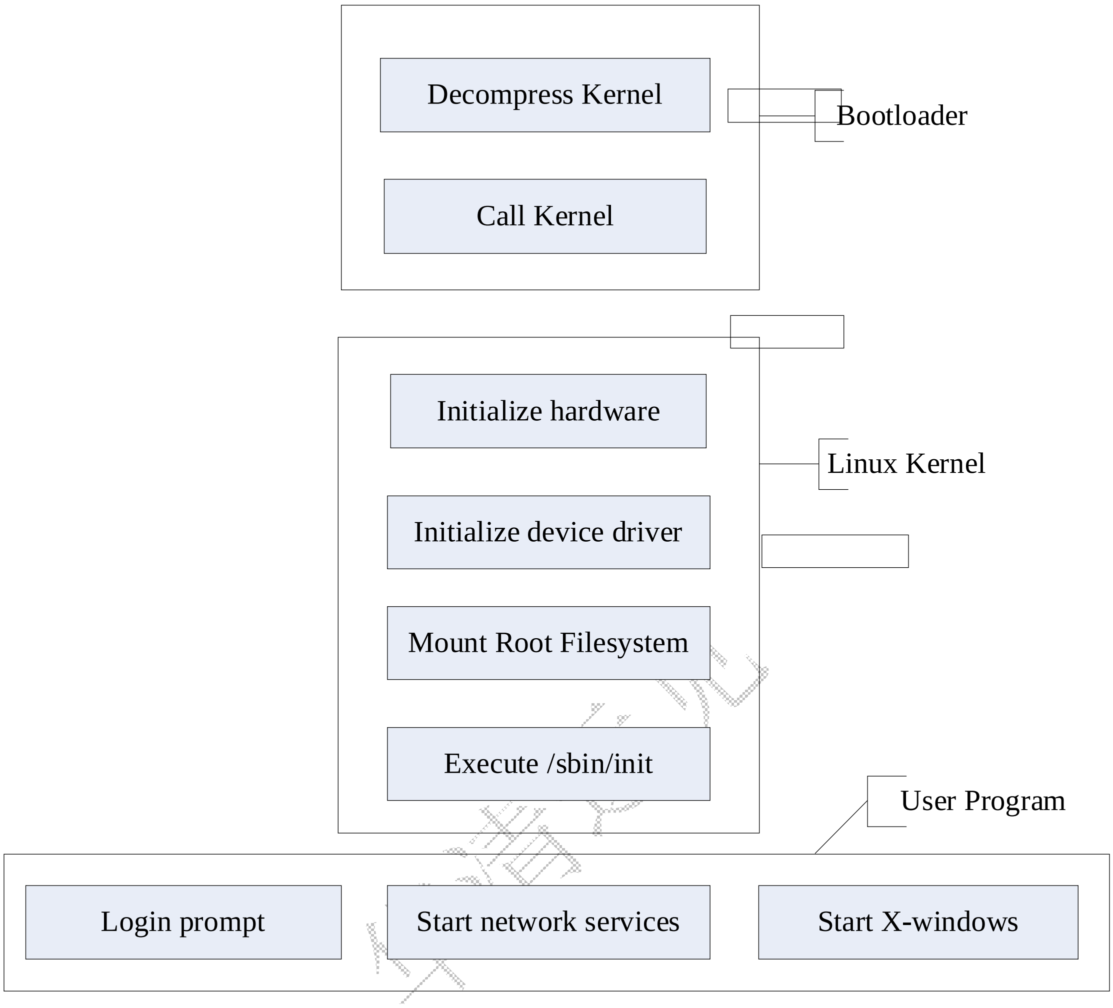


图 4.8 Linux 系统启动过程

第一阶段是目标板硬件初始化，解压内核映像，再跳转到内核映像入口。这部分的工作一般由目标板的引导程序和内核映像的自引导程序完成。不同体系结构的目标板引导的方式和程序都有差异。

第二阶段是内核的初始化，初始化设备驱动，挂接根文件系统。这里是 Linux 内核通用的启动函数入口。所有体系结构的目标板都顺序调用统一的函数，尽管有些函数的代码实现是跟体系结构相关的。

第三阶段是执行用户空间的 init 程序，完成系统初始化、启动相关服务和管理用户登录等工作。这个阶段可以提供给用户交互界面，例如：Shell 命令行或者图形化的窗口界面。也可以自动执行应用程序。

在 Linux 系统启动过程中，有两个关键点。一个是内核映像的解压启动；另一个是根文件系统的挂接。

4.4.2 内核解压启动

目标板处理器上电或者复位后，首先执行引导程序（Bootloader），初始化内存等硬件，然后把压缩的内核映像加载到内存中，最后跳转到内核映像入口执行。这样就把控制权完全交给内核映像了。

接下来内核映像继续执行，完成自解压或者重定位，然后跳转到解压后的内核代码入口。这部分主要是 Linux 内核的自引导程序，又叫作 Linux bootloader，包含在内核源代码中。这部分引导代码相对简单，不可能替代目标板上的 Bootloader。

目标板的 Bootloader 具有加载内核映像的功能。在嵌入式 Linux 开发中，经常用到网络加载的方式，就是通过 TFTP 协议把内核映像加载到目标板内存。那么目标板的 Bootloader 还应该能够驱动网络接口，配置 IP 地址。不同的 Bootloader 还有一系列命令进行配置。对于 U-boot 的使用和代码分析请参考第 6 章。

这里以 ARM 开发板的 U-boot 为例说明网络加载启动内核映像。

```
U-Boot 1.1.2 (Dec 25 2005 - 14:47:21)
U-Boot code: 33F80000 -> 33F98680  BSS: -> 33F9C7C0
RAM Configuration:
Bank #0: 30000000 64 MB
Flash:  2 MB
*** Warning - bad CRC, using default environment
In:    serial
Out:   serial
Err:   serial
=> tftp 30008000 zImage        #加载压缩的内核映像 zImage 到内存 0x30008000
TFTP from server 192.168.1.1; our IP address is 192.168.1.100
Filename 'zImage'.
Load address: 0x30008000
Loading: #################################################################
done
Bytes transferred = 1048584 (100008 hex)
=> go 30008000                              #跳转到压缩的内核映像入口
## Starting application at 0x30008000 ...   #内核映像自解压启动
Uncompressing Linux.....................................................
 done, booting the kernel.
version 2.6.14 (root@newasus) (gcc version 3.3.2) #5 Sun Dec 25 15:40:42 5CPU:
ARM920Tid(wb) [41129200] revision 0 (ARMv4T)          #开始执行内核初始化函数
Machine: SMDK2410
Warning: bad configuration page, trying to continue
Memory policy: ECC disabled, Data cache writeback
```

```
    CPU S3C2410A (id 0x32410002)
    S3C2410: core 200.000 MHz, memory 100.000 MHz, peripheral 50.000 MHz
    S3C2410 Clocks, (c) 2004 Simtec Electronics
    CLOCK: Slow mode (1.500 MHz), fast, MPLL on, UPLL on
    CPU0: D VIVT write-back cache
    CPU0: I cache: 16384 bytes, associativity 64, 32 byte lines, 8 sets
    CPU0: D cache: 16384 bytes, associativity 64, 32 byte lines, 8 sets
    Built 1 zonelists
    Kernel command line: root=/dev/nfs nfsroot=192.168.1.1:/usr/local/arm/3.3.2/
rootfs
    ……
```

这样内核就在目标板上启动起来了。

4.4.3　挂接根文件系统

因为文件和应用程序都要存储在文件系统中，所以 Linux 离不开文件系统。在内核启动到最后，必须挂接一个根文件系统。从文件系统的目录下找到 init 程序，启动 init 进程。

在交叉开发环境中，通常采用 NFS 文件系统。在内核启动过程可以挂接 NFS 根文件系统。这种方式将极大地方便嵌入式 Linux 交叉开发，在第 4.5 节可以很好地体会到这一点。

要使目标板挂接 NFS 根文件系统，需要做两方面的工作。一方面是在主机端配置相应的网络服务；另一个方面就是配置目标板的内核选项。

在第 4.3 节中已经详细说明了 TFTP、DHCP 和 NFS 服务的配置，有关主机端的网络服务配置可以参考。这里看一看还需要配置哪些内核选项。

Linux 内核要挂接 NFS 根文件系统，必须具备以下条件。

（1）以太网接口驱动正常

这需要配置相应的网络驱动程序。10/100M 以太网接口的驱动一般在菜单项“Network device support”下。

（2）配置内核启动命令行参数

实现上述功能，可以通过 DHCP 服务动态配置；也可以通过内核命令行参数指定。

配置内核启动命令行参数缺省值的菜单项“Default kernel command line string”。命令行格式如下。

```
    root=/dev/nfs rw nfsroot=<nfs_server>:<root_path> ip=<target_ip>
```

<target_ip>是为目标板指定的 IP 地址；

<nfs_server>是指定 NFS 服务器的 IP；

<root_path>是要挂接的 NFS 服务器的目录；

root=/dev/nfs 则指定要挂接 NFS 根文件系统；

rw 表示按照可读/写属性挂接。

（3）配置内核挂接 NFS 根文件系统

要使内核挂接 NFS 根文件系统，首先要支持网络协议配置选项，再选择 NFS 文件系统的支持，然后选择 NFS 为根文件系统。

配置编译完内核，下载到目标板上启动。注意要先把开发主机端的服务启动起来。如果目标板的 IP 地址和 NFS 网络路径需要通过 DHCPD 服务获取，可能需要根据目标板以太网接口的 MAC 地址修改 dhcpd.conf 配置文件，并重新启动。

网络根文件系统挂接成功以后，目标板就可以登录到 Shell，执行应用程序了。这样，交叉开发环境就建立起来了。

4.5 应用程序的远程交叉调试

4.5.1 交叉调试的模型

Linux 下 GCC 编译的程序都是采用 GDB 调试的。对于本地调试，要调试的程序和 GDB 运行在同一台主机上。如果目标板没有 gdb 的调试程序，或者没有 gdb 前端的图形化调试界面，像以前那样本地调试就不大可能。对于本地调试，交叉调试的 gdb 运行在开发主机上，而应用程序运行在目标板上。

在目标板上，通过 gdbserver 控制要调试的程序执行，同时与主机的 gdb 远程通讯。可以实现交叉调试的功能。这样，gdb 交叉调试运行在主机端，应用程序运行在目标板端。交叉调试模型如图 4.9 所示。

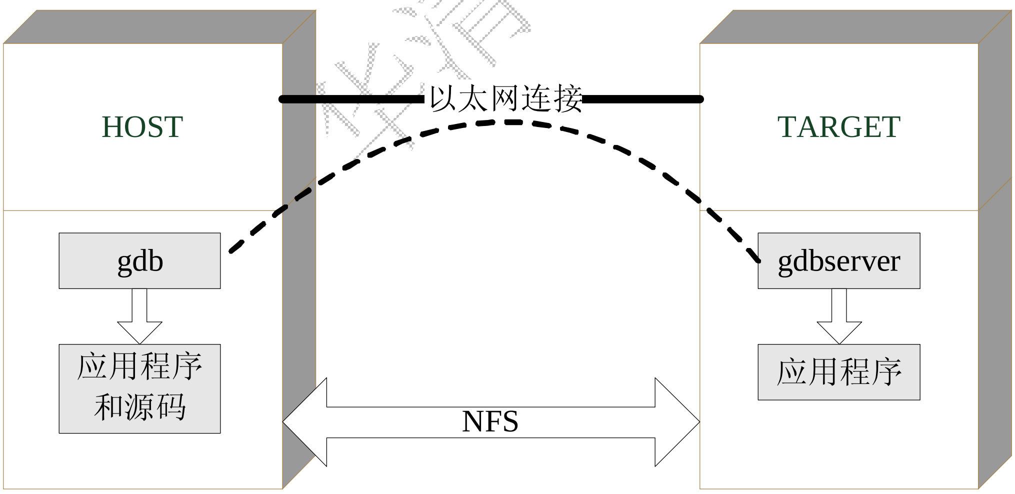


图 4.9 交叉调试模型

4.5.2 交叉调试程序实例

举一个简单的例子，来说明交叉调试的过程。

1．交叉编译

在开发主机上编辑一个 C 程序，再交叉编译，然后在目标板上执行。

（1）在主机上编辑 hello.c 程序

```
# include <stdio.h>
main(int argc, char **argv)
{
     int i;
     for ( i=0; i<3; i++)
     {
            printf("Hello i=%d\n", i);
     }
     return 0;
}
```

（2）交叉编译

```
$ arm-linux-gcc -o hello hello.c
```

（3）把可执行程序复制到 NFS 输出的目录下面

```
$ cp hello /usr/local/arm/3.3.2/rootfs
```

（4）这时在目标板端也可以访问到同样的程序，执行程序

```
# ./hello
hello i=1
hello i=2
hello i=3
```

这是一个简单的程序。如果工程包含多个文件，最好使用 Makefile 来管理编译。

2．交叉调试

接下来，还用这个程序，说明交叉调试应用程序过程。交叉调试需要目标板文件系统中必须有 gdbserver 工具。gdbserver 负责与远程的 gdb 远程通讯并且控制本地的应用程序执行。

（1）编译程序的时候，需要添加-g 编译选项，使程序包含调试信息

```
$ arm-linux-gcc  -g  –o  hello  hello.c
```

在主机上要调试的程序必须是带调试信息可执行程序；在目标板上执行的程序则可以使用一个精简过的可执行程序。

（2）在目标板上，启动 gdbserver，控制程序执行

```
# gdbserver <host>:2345 hello
```

<host>是主机名称或者 IP 地址。

2345 是网络端口号，服务器在这个端口上等待客户端的连接，这个值可以是任何目标板上可用的端口号。

hello 是调试程序名，还可以添加程序运行的参数。

控制台输出类似下面的显示。

```
Process hello created; pid = 38
```

（3）在主机端，启动 DDD 和 gdb 调试程序

```
$ ddd --debugger arm-linux-gdb hello
```

（4）在 DDD 下窗口的 GDB 控制台下，建立连接

```
(gdb)target remote <target>:2345
```

<target>是目标板 IP 地址，2345 是端口号，对应 gdbserver 启动时使用的端口号。连接成功，就可以使用 GDB 的命令调试了。

```
Remote debugging using 192.168.1.1:2345
```

（5）设置断点，执行到断点

在 main 函数设置断点。单击工具条上的 cont 执行到断点。按照下列命令行的方式同样可以进行调试。

```
(gdb)b main
(gdb)c
```

继续执行到断点，然后就可以单步执行或者设置更多断点调试了。

“黑色经典”系列之《嵌入式 Linux 系统开发技术详解——基于 ARM》

第 5 章 交叉开发工具链

本章目标

本章介绍编译生成 GNU 工具链的基本步骤。通过学习本章内容可以使读者理解交叉工具链的来源，并且体会到生成和维护工具链的复杂性。

- 工具软件的来源
- 制作交叉编译器
- 制作交叉调试器

5.1　工具链软件

Linux 软件从一开始就使用 GNU 的工具链。这些 GNU 的工具和软件都是开放源码的，可以免费下载源码编译。但是并不能以为任何一个版本拿来都能用，各种软件包存在版本匹配问题，并且不同版本都有一些补丁。

一套完善的工具链对于嵌入式 Linux 开发非常重要。发行版的 Linux 都会包含一整套工具链。工具链的维护和升级是 Linux 公司（特别是嵌入式 Linux 公司）非常重要的一项工作。

5.1.1　相关软件工程

GNU 的工具链源码包可以从 GNU 网站 http://www.gnu.org 或者镜像下载。这个站点有很多 GNU 软件，其中 Linux 使用的工具链软件是：BINTUTILS、GCC、GLIBC 和 GDB。

通过这些软件包，可以生成 gcc、g++、ar、as、ld 等编译链接工具，还可以生成 glibc 库和 gdb 调试器。这些编程工具的使用在第 3 章有详细说明。对于交叉开发的工具链来说，在文件名字上加了一个前缀，用来区别本地的工具链。例如：arm-linux-gcc，除了体系结构相关的编译选项以外，它的使用方法与 Linux 主机上的 GCC 相同。所以 Linux 编程技术对于嵌入式 Linux 同样适用。

交叉开发工具链就是为了编译、链接、处理和调试跨平台体系结构的程序代码。在 X86 的 Linux 主机上，除了编译生成 ARM MIPS PowerPC 等体系结构的程序，还可以为 X86 不同版本的 Linux 开发程序。例如：为了维护不同版本的 X86 目标机，可以在 Redhat Linux 9 的主机上通过交叉编译的方式开发。

下面介绍一下这些软件工程的一些特点。

BINUTILS 是二进制程序处理工具，包括连接器、汇编器等目标程序处理的工具。

GCC（GNU Compiler Collection）是编译器，不但能够支持 C/C++语言的编译，而且能够支持 FORTRAN JAVA ADA 等编程语言。不过，一般不需要配置其他语言的选项，也可以避免编译其他语言功能而导致的错误。对于 C/C++语言的完整支持，需要支持 glibc 库。

GLIBC 是应用程序编程的函数库软件包，可以编译生成静态库和共享库。完整的 GCC 需要支持 glibc。

GDB 是调试工具，可以读取可执行程序中的符号表，对程序进行源码调试。

5.1.2　软件版本的匹配

1．Crosstool

Crosstool 软件实际上是一套脚本，用于编译和测试大多数体系结构的各种 GCC 和 glibc 的版本组合。当然，前提是 glibc 能够支持这些体系结构，它还为工具链源码包提供了补丁。从 Crosstool 网站上，可以下载到这些编译脚本、补丁和文档。

Crosstool 包含了体系结构和 gcc、glibc 各种组合配置的最小补丁。Crosstool 测试支持范围如表 5.1 所示。

表 5.1　　crosstool 测试支持范围

处理器体系结构	alpha, arm, i686, ia64, mips, powerpc, powerpc64, sh4, sparc, sparc64, s390, x86_64
gcc 版本	gcc-2.95.3 ... gcc-4.0.0
glibc 版本	glibc-2.1.3 ... glibc-2.3.5

crosstool 的新版本会不断扩大测试范围。不妨下载 crosstool-0.38 版本，看看软件中大堆的脚本和补丁。

```
$ wget  -c  http://www.kegel.com/crosstool/crosstool-0.38.tar.gz
$ tar –zxvf crosstool-0.38.tar.gz
$ cd crosstool-0.38
```

顶层目录下有很多*.sh 脚本和*.dat 配置文件，这是对于各种体系结构和工具版本进行编译测试的脚本。例如：all.sh、demo-arm.sh、arm.dat、gcc-4.1-20050709-glibc-2.3.2-hdrs-2.6.11.2.dat 等。

因为文件确实太多，我们就不一一分析这些脚本了，只把几个目录归纳说明，如表 5.2 所列。

表 5.2　　crosstool 目录说明

目　　录	说　　明
buildlogs	编译各种各样版本的日志文件
contrib.	为社区贡献的补丁
doc	开发使用文档，初次接触的时候可以看一看
patches	各种版本工具的补丁，包含 binutils、gcc、glibc 各种版本的补丁
summaries	几种体系结构的测试结果总结

Crosstool 的维护者是 Dan Kegel，可以到 http://www.kegel/crosstool/查看最新信息。

2．LFS（Linux From Scratch）

顾名思义，LFS 就是要指导人们从头开始制作 Linux 系统。它提供详细的操作步骤，从源代码开始，一步一步地编译出自己的 Linux 系统。

LFS 最大的优点是可以按照自己的喜好和需要定制自己的系统。它可以帮助人们了解 Linux 系统从头到脚到底是怎么工作的，打造一个 LFS 系统的过程，把 Linux 内部各个部分如何协调工作以及互相的依赖关系都展示出来。

LFS 第 2 个优点是可以从更大的程度上控制开发者自己的系统，而不依赖于别人打造的工具。开发者成为了 Linux 系统每个部分的操纵者，比如目录的分配和起动脚本，开发者还可以了解每一个程序是做什么的，装在哪里，如何安装。

LFS 第 3 个优点是你可以建立一个很小的 Linux 系统。在安装一般的 Linux 发行版的时候，最后需要较大的硬盘空间，其中安装了一些可能并不需要的程序。可以建立一个小体积的嵌入式 LFS 系统，成功地把一个系统缩减到了 8M，并且可以支持 Apache 网络服务器。 进一步的简化可以把体积压缩到 5M 以下。

LFS 第 4 个优点是系统安全性。由于整个系统都是自己定制的，可以在编译系统源码的时候加进任何一个想要的安全补丁，就不用去等别人打过补丁编译好的二进制包了。

这样的优点举不胜举，这只是其中比较突出的特点。随着 LFS 经验逐渐增加，你会在自己身上发现知识所带来的力量。

LFS 工程又分成几个项目，每个项目的内容如表 5.3 所示。其中 LFS 是最基本的文档，其他项目的文档或者软件是进一步开发的工具。对于嵌入式 Linux 系统开发来说，LFS 和 CLFS 都是很好的文档。

表 5.3 LFS 文档项目

文 档 名 称	说 明
LFS（Linux From Scratch）	最主要的文档，它是其他项目的基础
BLFS（Beyond Linux From Scratch）	扩展根据 LFS 制作的系统功能
ALFS（Automated Linux From Scratch）	提供 LFS 等自动管理和编译的工具
CLFS（Cross Linux From Scratch）	提供各种体系结构的交叉编译方法
HLFS（Hardened Linux From Scratch）	关注系统的安全性
Hints	提高系统性能的说明文档，LFS 或者 BLFS 没有包含的部分
LiveCD	LFS 开发主机光盘和修复盘
Patches	构建 LFS 主要的补丁

最新版本的 LFS 文档可以从 LFS 站点 http://www.linuxfromscratch.org 下载。

LFS 本文档列出了构建 LFS 需要的所有软件包索引。每个软件包都列出了软件包的主页、使用手册和相关资料的链接地址。LFS 书包含了构建 LFS 的所有东西，有充足的信息，并且还有一些培训资料。因此，可以自己解决使用过程中的问题。

这里是开始理解曾经安装过的软件的好地方，但是要自己动手编译这个系统。这个自己动手构建的 Linux 系统是在 X86 的平台上的。对于嵌入式 Linux 系统开发，可以重点学习一些工具软件和系统软件包。

CLFS（Cross Linux From Scratch）告诉读者如何制作一个交叉编译器以及必需的工具。基于各种体系结构构建一个基本的系统。例如：可以在 X86 系统上制作 Sparc 工具链，并且利用这个工具链从源码编译 Linux 系统。

LFS 文档由 Matthew Burgess 编写维护，其他的项目有更多的人参与。

3．常用版本

Binutils、GCC 和 glibc 的版本匹配是个大麻烦。应该说，越新的版本功能越强大，但是最新版本有可能存在 BUG，这就需要不断地测试修正。

对于 GCC 的版本，2.95.x 曾经统治了 Linux 2.4 内核时代，它表现得极为稳定。现在 GCC2.95.3 版本已经过时了，Linux 2.6 内核需要更高的工具链版本支持。Linux 2.6 内核最好使用 GCC 3.3 以上版本。

对于 glibc 版本，还要跟 Linux 内核的版本号匹配。在编译 glibc 时，要用到 Linux 内核头文件，在内核源码的 include 目录下。如果发现有变量没有定义而导致编译失败，就改变内核版本号。如果没有完全的把握保证内核修改完全了，就不要动内核，而应该把 Linux 内核的版本号降低或升高，以适应 glibc 版本。如果选择的 glibc 的版本号低于 2.2，还要下载一个 glibc-crypt 的软件包，例如 glibc-crypt-2.1.tar.gz。解压到 glibc 源码树中。

对于 binutils 版本，可以尽量使用新的版本，很多工具是辅助 GCC 编译的功能，问题相对较少。

2.4 内核和 2.6 内核的工具链版本的基本组合见表 5.4，这些是在 ARM 平台上测试过的。新的处理器或者体系结构都要求使用更高的版本才能够支持。

表 5.4　　ARMV4T 平台工具链常用版本

工具链版本	Linux 2.4.x	Linux 2.6.x
binutils	2.14	2.14
gcc	2.95.3	3.3.2
glibc	2.2.5	2.2.5
glibc-threads	2.2.5	2.2.5
gdb	5.3	6.0

5.1.3　工具链制作流程

由于工具链是几个软件包组合编译生成的，所以每个软件包的编译并不是独立的，相互存在依赖关系。

一个关键的问题是完整的 GCC 编译器是依赖于 glibc 的，而 glibc 需要对应体系结构的 GCC 来编译。这怎么解决呢？先编译一个辅助的 GCC 编译器（Bootstrap compiler），用来编译 glibc，然后再重新编译完整的 GCC 编译器。

图 5.1 是编译工具链的流程图。可以把这个过程划分为 5 个步骤。

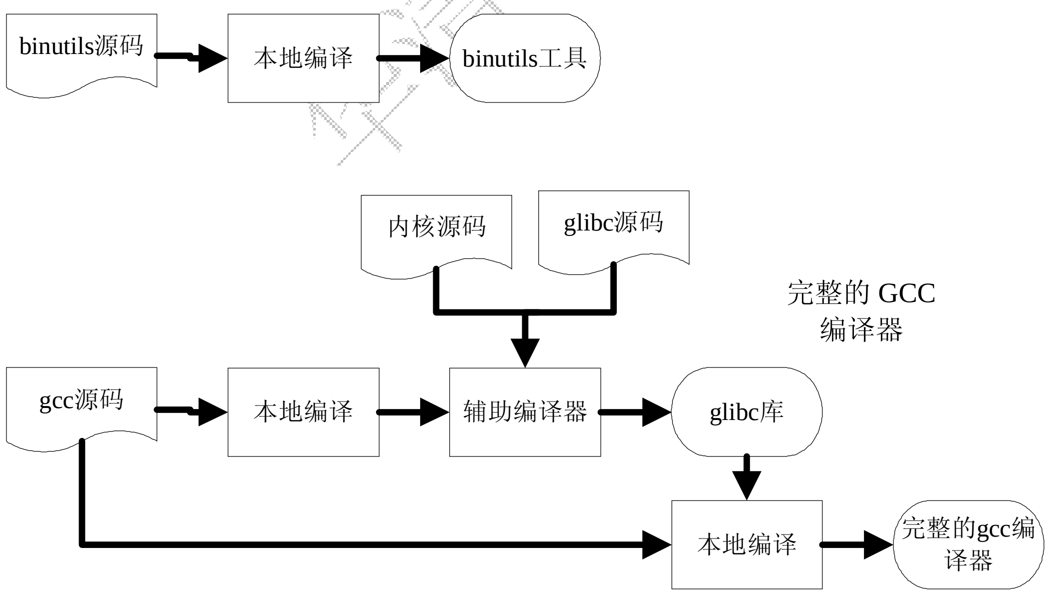


图 5.1　交叉工具链编译流程图

① 做好准备工作。下载工具源码包和补丁，准备内核头文件，创建工作目录等。

② 编译 binutils。这个软件包的编译一般很顺利，不会出现什么问题。

③ 编译辅助编译器。这一步使用简化配置，编译通常也很顺利。

④ 编译 glibc 库。这里要使用交叉编译工具链，例如：arm-linux-gcc 等。

⑤ 编译生成完整的 GCC 编译器。重新配置 GCC 功能，使其支持 C、C++等语言。

注意在配置编译之前，首先要查看所有补丁，把需要的补丁打进去。有的补丁可以解决编译过程的语法错误，有的补丁专门修正对某个体系结构的支持，有时还要手工地修改一些文件内容。其中，后两步编译过程最麻烦，编译一般会出现一些错误。

编译 binutils、gcc、glibc 是最基本的工具链。如果要调试程序，还离不开调试器 gdb。相对来说，gdb 的编译很简单，它也应该作为工具链配套提供。最后还要编译安装交叉调试器。

本章的内容着重描述制作过程，而且举例说明的版本已经顺利编译通过。但是在使用过程中，很有可能发现工具链在某个方面存在问题，甚至严重的 BUG，这就需要不断地修正 BUG，它和 Linux 内核一样是不断发展提高的。

5.2 制作交叉编译器

5.2.1 准备编译环境

我们来自己动手编译这套交叉开发工具链。选择 GCC-3.3.2 的版本，在 ARM 体系结构平台上，这个版本既能支持 Linux 2.4 内核开发，又能够支持 Linux 2.6 内核开发。

首先准备编译环境。创建一个工作目录~/crosstool，把下载的源码包放到~/crosstool/source 目录下。

接下来下载相关软件包。这些软件包从 GNU 或者 FSF 官方站点上都可以下载，也可以从其他嵌入式 Linux 网站下载。例如：ARM Linux 官方站点就提供 binutils、gcc 和 glibc 的源代码包，但是一般没有包含 gdb 的软件包。

这些软件包都很大，可以考虑使用下载工具。Linux 下的 wget 就很好，可以支持 FTP 和 HTTP，还支持断点续传。下载命令如下。

```
$ wget  -c  ftp://ftp.gnu.org/gnu/binutils/binutils-2.14.tar.bz2
$ wget  -c  ftp://ftp.gnu.org/gnu/gcc/gcc-3.3.2.tar.bz2
$ wget  -c  ftp://ftp.gnu.org/gnu/glibc/glibc-2.2.5.tar.bz2
$ wget  -c  ftp://ftp.gnu.org/gnu/glibc/glibc-linuxthreads-2.2.5.tar.bz2
$ wget  -c  ftp://ftp.gnu.org/gnu/gdb/gdb-6.0.tar.bz2
```

不要忘了找找工具链的补丁。最好下载最新版本的 crosstool 软件包，从中可以找到一些有用的东西。对于编译的语法错误，已经有相应的补丁修正。当然，我们也可以手工修改。

还要准备内核头文件目录。从内核源码中把 include 目录复制过来。

准备好工作区，创建如下目录。

```
~/crosstool/source
~/crosstool/buildlogs
```

```
~/crosstool/patches
~/crosstool/linux-2.6.x
~/crosstool/xxxxxx/build-arm-linux
```

xxxxxx 代表 binutils、glibc、gcc、gdb 的源代码目录。在这些目录下创建 build-arm-linux 目录，作为临时工作目录。配置编译过程生成 Makefile、目标文件、临时文件等，都存放在 build-arm-linux 目录下。这样所有编译的内容都在独立的一个目录下，甚至可以为多种体系结构建立各自的编译目录。

编译的过程可能会出错，导致编译过程无法继续进行。详细分析出错信息，有助于解决源码中的语法错误。

那么如何保存配置编译过程的信息？这些信息量很大，都可能超出 Shell 向上翻滚查看的范围。最好是把编译过程的信息保存成日志文件，方便后面的分析。

举例说明保存编译信息的行命令，它把 make 过程打印的所有信息都保存在 xxx.log 中。

```
$ make   2>&1 | tee xxx.log
```

这条命令是编译并保存打印信息。在 Linux Shell 的设备定义中，“0”表示标准输入，“1”表示标准输出，“2”表示标准出错信息输出。2>&1 表示把 2 设备的信息重定向到 1 设备；“|”是管道符号，把标准输出的信息直接传递给后面的命令；tee 是创建文件并保存信息的工具；xxx.log 是文件名。

这种管道的用法在 Linux Shell 命令中使用非常普遍。下面的编译过程中都可以使用这个方法，生成日志文件，保存到 buildlogs 目录下。不过为了看起来简洁，下面的操作步骤省略了这一项。

5.2.2 编译 binutils

按照下列步骤编译 binutils。

```
$ tar -jxf ./source/binutils-2.14.tar.bz2
$ cd binutils-2.14
$ mkdir build-arm-linux
$ cd build-arm-linux
$ ../configure --target=arm-linux --prefix=/usr/local/arm/3.3.2
$ make
$ make install
```

解压源码包，并在 binutils-2.14 源码目录中创建 build-arm-linux 目录，作为配置编译工作目录。

--target 指定交叉工具的目标板体系结构，所有运行在主机上的交叉工具都要配置这个选项。

--prefix 指定路径前缀，编译完成以后，将安装到这个目录下。使用交叉工具链也是与路径有关系的。

编译过程简单顺利，编译出来的文件或者工具全部安装到/usr/local/arm/3.3.2/目录下。

我们需要的开发工具都在 bin/目录下。这时查看一下 bin 目录下的文件，可以得到下列

交叉开发工具。

```
$ ls /usr/local/arm/3.3.2/bin
arm-linux-addr2line
arm-linux-ar
arm-linux-as
arm-linux-c++filt
arm-linux-ld
arm-linux-nm
arm-linux-objcopy
arm-linux-objdump
arm-linux-ranlib
arm-linux-readelf
arm-linux-size
arm-linux-strings
arm-linux-strip
```

这些 GNU 开发工具是都带 arm-linux 前缀。这些工具的用法跟主机本地的工具是相同的，只是处理的二进制程序体系结构不同。在开发主机上，如果是为目标板平台编译可执行程序，一定要用交叉开发工具。

记住把新生成的工具添加到环境变量 PATH 中，可以在 Linux 启动脚本添加：

```
export PATH=$PATH:/usr/local/arm/3.3.2/bin
```

5.2.3 编译 GCC 的辅助编译器

按照下列步骤编译辅助编译器。

（1）解压源码包

```
$ tar -jxf ./source/gcc-3.3.2-tar.bz2
$ cd gcc-3.3.2
```

（2）配置 t-linux 文件

因为现在还没有 glibc 的支持，所以需要配置一些简单选项。对于 arm-linux 工具，可以修改 gcc/config/arm/t-linux 配置文件。

编辑 t-linux，在文件末尾添加如下 2 行。

```
#  gcc/config/arm/t-linux
……
         TARGET_LIBGCC2_CFLAGS += -Dinhibit_libc -D__gthr_posix_h
         T_CFLAGS = -Dinhibit_libc -D__gthr_posix_h
```

-Dinhibit_libc 选项意思是禁止使用 libc，因为现在还没有为目标板编译出 glibc 库。

（3）配置

```
$ mkdir build-arm-linux
$ cd build-arm-linux
$ ../configure --target=arm-linux \
      --prefix=/usr/local/arm/3.3.2 \
      --with-headers=~/crosstool/linux-2.6.x/include \
      --disable-shared --disable-threads --enable-languages="c"
```

--target 指定交叉工具的目标板体系结构是 arm-linux。

--prefix 指定安装路径为/usr/local/arm/3.3.2。

--with-headers 指定内核头文件所在路径为~/crosstool/linux-2.6.x/include。

--disable-shared 选项指定不使用共享库，这样就不依赖于 glibc 了。

--disable-threads 选项指定不使用线程，也就不使用线程库了。

--enable-languages 指定仅支持 C 语言。

（4）编译

```
$ make
```

这个编译过程会比较顺利，但是要花的时间稍微长一些。如果编译出错，可能是由于 t-linux 配置的原因。可以从网上再搜索一些别人的例子，比较一下。

（5）安装

```
$ make install
```

生成的文件也安装到/usr/local/arm/3.3.2 目录下。主要是在 bin/目录下添加 arm-linux-gcc 工具。不过现在是辅助的编译器，编译 glibc 的时候要使用。

5.2.4　编译生成 glibc 库

按照下列步骤编译 glibc。

（1）解压源码包

```
$ tar xvzf ~/crosstool/source/glibc-2.2.5.tar.gz
$ cd glibc-2.2.5
$ tar xvzf ~/crosstool/source/glibc-linuxthreads-2.2.5.tar.gz
```

（2）修正 BUG

glibc 的源码中有一些 BUG 需要修正，否则编译过程肯定会频频出错。可以通过各种方式来修改，例如：使用脚本语言、补丁工具或者编辑器 vim。

下面说明一下要修改的文件以及修改的内容，“+”号表示添加这一行，“–”号表示删除这一行。

sysdeps/unix/sysv/linux/configure：参照 i386 的选项，在 i386 上面添加 arm 的选项。

```
+  arm*)
+    libc_cv_gcc_unwind_find_fde=yes
+    arch_minimum_kernel=2.0.10
```

```
+    ;;
   i386*)
     libc_cv_gcc_unwind_find_fde=yes
     arch_minimum_kernel=2.0.10
```

sysdeps/unix/sysv/linux/arm/errlist.c：删除 2 行 weak_alias()，添加 1 行 strong_alias()。

```
-weak_alias (__old_sys_nerr, _old_sys_nerr)
 compat_symbol (libc, __old_sys_nerr, _sys_nerr, GLIBC_2_0);
 compat_symbol (libc, _old_sys_nerr, sys_nerr, GLIBC_2_0);
-weak_alias (__old_sys_errlist, _old_sys_errlist);
+strong_alias (__old_sys_errlist, _old_sys_errlist);
 compat_symbol (libc, __old_sys_errlist, _sys_errlist, GLIBC_2_0);
 compat_symbol (libc, _old_sys_errlist, sys_errlist, GLIBC_2_0);
 #endif
```

stdio-common/sprintf.c：sprintf()函数添加变参。

```
int
-sprintf (s, format)
-       char *s;
-       const char *format;
+sprintf (char *s, const char *format, ...)
 {
   va_list arg;
   int done;
```

stdio-common/sscanf.c：sscanf()函数添加变参。

```
int
–sscanf (s, format)
–      const char *s;
–      const char *format;
+sscanf (const char *s, const char *format, ...)
 {
   va_list arg;
   int done;
```

sysdeps/unix/sysv/linux/arm/sysdep.h：从列表中删除“a1”。

```
        asm volatile ("swi %1  @ syscall " #name   \
               : "=r" (_a1)                    \
               : "i" (SYS_ify(name)) ASM_ARGS_##nr   \
```

```
-               : "a1", "memory");                \
+               : "memory");                \
        _sys_result = _a1;                  \
      }                                     \
      if (_sys_result >= (unsigned int) -4095)       \
```

sysdeps/arm/dl-machine.h：在宏定义的汇编语句每一行末尾添加“\n\”，涉及的汇编语句很多，要全部加上。这里只以 CALL_ROUTINE 宏定义举例说明，另外还要修改 RTLD_START 和 ELF_MACHINE_RUNTIME_TRAMPOLINE 两个宏定义中的汇编语句。

```
   and then redirect to the address it returns.  */
   // macro for handling PIC situation...
 #ifdef PIC
-#define CALL_ROUTINE(x) " ldr sl,0f
-     add  sl, pc, sl
-1:   ldr  r2, 2f
-     mov  lr, pc
-     add  pc, sl, r2
-     b   3f
-0:   .word_GLOBAL_OFFSET_TABLE_ - 1b - 4
-2:   .word " #x "(GOTOFF)
+#define CALL_ROUTINE(x) "\
+    ldr sl,0f\n\
+    add    sl, pc, sl\n\
+1:  ldrr2, 2f\n\
+    movlr, pc\n\
+    addpc, sl, r2\n\
+    b      3f\n\
+0:  .word  _GLOBAL_OFFSET_TABLE_ - 1b - 4\n\
+2:  .word " #x "(GOTOFF)\n\
 3: "
 #else
 #define CALL_ROUTINE(x) " bl " #x
```

sysdeps/unix/sysv/linux/arm/ioperm.c：把 BUS_ISA 替换成 CTL_BUS_ISA。

```
@@ -100,8 +106,8 @@
 {
   char systype[256];
   int i, n;
-static int iobase_name[] = { CTL_BUS, BUS_ISA, BUS_ISA_PORT_BASE };
```

```
– static int ioshift_name[] = { CTL_BUS, BUS_ISA, BUS_ISA_PORT_SHIFT };
+ static int iobase_name[] = { CTL_BUS, CTL_BUS_ISA, BUS_ISA_PORT_BASE };
+ static int ioshift_name[] = { CTL_BUS, CTL_BUS_ISA, BUS_ISA_PORT_SHIFT };
```

config.make.in：把"slibdir=@...@"替换成"slibdir=@libdir@"。

最后，有些程序的函数参数"_thread"名称都替换成"_thr"。相关文件如下。

linuxthreads/sysdeps/pthread/pthread.h

linuxthreads/internals.h

linuxthreads/sysdeps/unix/sysv/linux/bits/sigthread.h

（3）配置

```
$ mkdir build-arm-linux
$ cd build-arm-linux
$ CC=arm-linux-gcc \
AS=arm-linux-as \
LD=arm-linux-ld \
../configure --host=arm-linux \
      --with-headers=~/crosstool/linux-2.6.x/include \
      --enable-add-ons=linuxthreads --enable-shared \
      --prefix=/usr/local/arm/3.3.2/arm-linux
```

编译 glibc 时，要使用交叉编译工具，因此定义环境变量 CC AS LD 为交叉工具链。

--host 指定目标板的体系结构。

--with-headers 指定内核头文件路径为~/crosstool/linux-2.6.x/include。

--enable-add-ons=linuxthread 选项支持线程库。

--enable-shared 选项支持共享库。

--prefix 指定的是工具链目标板相关文件的目录：安装目录下的 arm-linux 目录。

（4）编译

```
$ make
```

如果编译过程出错，会大大影响编译速度。因此在修改 BUG 的时候要彻底，否则要花费更长的时间。

（5）安装

```
$ make install
```

结果在/usr/local/arm/3.3.2/arm-linux 目录下的 lib 等目录中，安装了 glibc 共享库等文件。

5.2.5 编译生成完整的 GCC 编译器

返回到 gcc-3.3.2 的源码目录下，按照下列步骤编译完整的 gcc 编译器。

（1）修改 t-linux

```
$ cd gcc-3.3.2
$ vi gcc/config/arm/t-linux
```

这里再修改 t-linux，把添加的 2 行-Dinhibit_libc 去掉。不能再禁止使用 libc，这里重新编译支持 libc 的完整的 GCC 编译器。

（2）清除临时文件

```
$ cd gcc-3.3.2/build-arm-linux/
$ make distclean
$ rm -rf ./*
```

把辅助编译器的配置清除掉，最后剩余的文件可以手工删除。

（3）配置

```
    $ ../configure --target=arm-linux \
            --prefix=/usr/local/arm/3.3.2 \
            --with-headers=~/crosstool/linux-2.6.x/include \
            --enable-threads=pthreads \
            --enable-shared \
            --enable-static \
            --enable-languages="c,c++"
```

--target 指定交叉工具的目标板体系结构是 arm-Linux。

--prefix 指定安装路径为/usr/local/arm/3.3.2。

--with-headers 指定内核头文件所在路径为~/crosstool/linux-2.6.x/include。

--enable-threads 选项指定使用线程库。

--enable-shared 选项指定使用共享库。

--enable-languages 指定支持 C 和 C++语言。如果这一项不设置，会把 Java 等语言的支持也编译进来，那么可能会由于编译其他不需要的功能而出现错误，不能顺利编译通过。

（4）编译

```
$ make
```

这一次编译的条件多一些，极有可能出现错误，一定细心配置。编译时间也比前面的过程长。

（5）安装

```
$ make install
```

编译的结果是得到完整的编译器 arm-linux-gcc 和 arm-linux-g++。到现在为止，完全可以使用这一套工具链进行交叉编译了。

5.3 制作交叉调试器

5.3.1 编译交叉调试器

对于交叉调试器，并不是工具链必需的工具。但是它是与工具链配套使用的，GDB 的调试能力和 BUG 的修正也因为版本的不同而不同。这里仅说明一下基本的编译过程。

按照下列步骤编译交叉调试器。

（1）解压源码包

```
$ tar -zxvf ./source/gdb-6.0-tar.gz
$ cd gdb-6.0
```

（2）配置

```
$ mkdir build-arm-linux
$ cd build-arm-linux
$ ../configure --target=arm-linux --prefix=/usr/local/arm/3.3.2
```

配置也很简单，只需要配置--target 和--prefix，指定目标板体系结构和安装路径即可。

（3）编译

```
$ make
```

耐心等待，一般不会出错。

（4）安装

```
$ make install
```

编译结果是在/usr/local/arm/3.3.2/bin 目录下得到 arm-linux-gdb 工具。

5.3.2 编译 gdbserver

根据第 4 章的交叉调试的例子，目标板还需要 gdbserver 工具为目标板交叉编译 gdbserver。Gdbserver 的源码在 gdb 源码树的 gdb/gdbserver 目录下。按照下列步骤配置编译。

```
$ cd gdb-6.0
$ cd gdb/gdbserver
$ chmod u+x configure
$ CC=arm-linux-gcc \
./configure --host=arm-linux
$ make
```

使用 arm-linux-gcc 交叉编译，配置--host 为 arm-linux。

编译结果生成 gdbserver 和 gdbreplay，这是目标板体系结构的可执行程序，复制到目标机文件系统中。

第 6 章　Bootloader

本章目标

本章介绍了 **Bootloader** 的概念和类型，重点讲解了 **U-Boot** 的开发调试和使用。通过学习 **U-Boot** 软件，可以使读者充分理解 **Bootloader** 的工作原理和代码实现。

- Bootloader 概况
- U-Boot 软件开发
- U-Boot 使用技巧

6.1 Bootloader

对于计算机系统来说，从开机上电到操作系统启动需要一个引导过程。嵌入式 Linux 系统同样离不开引导程序，这个引导程序就叫作 Bootloader。

6.1.1 Bootloader 介绍

Bootloader 是在操作系统运行之前执行的一段小程序。通过这段小程序，我们可以初始化硬件设备、建立内存空间的映射表，从而建立适当的系统软硬件环境，为最终调用操作系统内核做好准备。

对于嵌入式系统，Bootloader 是基于特定硬件平台来实现的。因此，几乎不可能为所有的嵌入式系统建立一个通用的 Bootloader，不同的处理器架构都有不同的 Bootloader。Bootloader 不但依赖于 CPU 的体系结构，而且依赖于嵌入式系统板级设备的配置。对于 2 块不同的嵌入式板而言，即使它们使用同一种处理器，要想让运行在一块板子上的 Bootloader 程序也能运行在另一块板子上，一般也都需要修改 Bootloader 的源程序。

反过来，大部分 Bootloader 仍然具有很多共性，某些 Bootloader 也能够支持多种体系结构的嵌入式系统。例如，U-Boot 就同时支持 PowerPC、ARM、MIPS 和 X86 等体系结构，支持的板子有上百种。通常，它们都能够自动从存储介质上启动，都能够引导操作系统启动，并且大部分都可以支持串口和以太网接口。

本章将对各种 Bootloader 总结分类，分析它们的共同特点。以 U-Boot 为例，详细讨论 Bootloader 的设计与实现。

6.1.2　Bootloader 的启动

Linux 系统是通过 Bootloader 引导启动的。一上电，就要执行 Bootloader 来初始化系统。可以通过第 4 章的 Linux 启动过程框图回顾一下。

系统加电或复位后，所有 CPU 都会从某个地址开始执行，这是由处理器设计决定的。比如，X86 的复位向量在高地址端，ARM 处理器在复位时从地址 0x00000000 取第一条指令。嵌入式系统的开发板都要把板上 ROM 或 Flash 映射到这个地址。因此，必须把 Bootloader 程序存储在相应的 Flash 位置。系统加电后，CPU 将首先执行它。

主机和目标机之间一般有串口可以连接，Bootloader 软件通常会通过串口来输入输出。例如：输出出错或者执行结果信息到串口终端，从串口终端读取用户控制命令等。

Bootloader 启动过程通常是多阶段的，这样既能提供复杂的功能，又有很好的可移植性。例如：从 Flash 启动的 Bootloader 多数是两阶段的启动过程。从后面 U-Boot 的内容可以详细分析这个特性。

大多数 Bootloader 都包含 2 种不同的操作模式：本地加载模式和远程下载模式。这 2 种操作模式的区别仅对于开发人员才有意义，也就是不同启动方式的使用。从最终用户的角度看，Bootloader 的作用就是用来加载操作系统，而并不存在所谓的本地加载模式与远程下载模式的区别。

因为 Bootloader 的主要功能是引导操作系统启动，所以我们详细讨论一下各种启动方式的特点。

1．网络启动方式

这种方式开发板不需要配置较大的存储介质，跟无盘工作站有点类似。但是使用这种启动方式之前，需要把 Bootloader 安装到板上的 EPROM 或者 Flash 中。Bootloader 通过以太网接口远程下载 Linux 内核映像或者文件系统。第 4 章介绍的交叉开发环境就是以网络启动方式建立的。这种方式对于嵌入式系统开发来说非常重要。

使用这种方式也有前提条件，就是目标板有串口、以太网接口或者其他连接方式。串口一般可以作为控制台，同时可以用来下载内核影像和 RAMDISK 文件系统。串口通信传输速率过低，不适合用来挂接 NFS 文件系统。所以以太网接口成为通用的互连设备，一般的开发板都可以配置 10M 以太网接口。

对于 PDA 等手持设备来说，以太网的 RJ-45 接口显得大了些，而 USB 接口，特别是 USB 的迷你接口，尺寸非常小。对于开发的嵌入式系统，可以把 USB 接口虚拟成以太网接口来通讯。这种方式在开发主机和开发板两端都需要驱动程序。

另外，还要在服务器上配置启动相关网络服务。Bootloader 下载文件一般都使用 TFTP 网络协议，还可以通过 DHCP 的方式动态配置 IP 地址。

DHCP/BOOTP 服务为 Bootloader 分配 IP 地址，配置网络参数，然后才能够支持网络传输功能。如果 Bootloader 可以直接设置网络参数，就可以不使用 DHCP。

TFTP 服务为 Bootloader 客户端提供文件下载功能，把内核映像和其他文件放在/tftpboot

目录下。这样 Bootloader 可以通过简单的 TFTP 协议远程下载内核映像到内存。如图 6.1 所示。

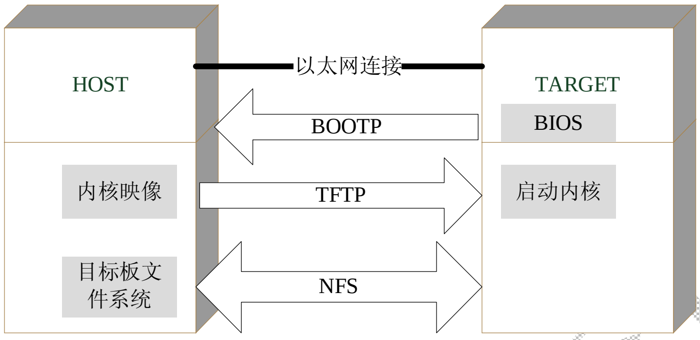


图 6.1 网络启动示意图

大部分引导程序都能够支持网络启动方式。例如：BIOS 的 PXE（Preboot Execution Environment）功能就是网络启动方式；U-Boot 也支持网络启动功能。

2．磁盘启动方式

传统的 Linux 系统运行在台式机或者服务器上，这些计算机一般都使用 BIOS 引导，并且使用磁盘作为存储介质。如果进入 BIOS 设置菜单，可以探测处理器、内存、硬盘等设备，可以设置 BIOS 从软盘、光盘或者某块硬盘启动。也就是说，BIOS 并不直接引导操作系统。那么在硬盘的主引导区，还需要一个 Bootloader。这个 Bootloader 可以从磁盘文件系统中把操作系统引导起来。

Linux 传统上是通过 LILO（LInux LOader）引导的，后来又出现了 GNU 的软件 GRUB（GRand Unified Bootloader）。这 2 种 Bootloader 广泛应用在 X86 的 Linux 系统上。你的开发主机可能就使用了其中一种，熟悉它们有助于配置多种系统引导功能。

LILO 软件工程是由 Werner Almesberger 创建，专门为引导 Linux 开发的。现在 LILO 的维护者是 John Coffman，最新版本下载站点：http://lilo.go.dyndns.org。LILO 有详细的文档，例如 LILO 套件中附带使用手册和参考手册。此外，还可以在 LDP 的“LILO mini-HOWTO”中找到 LILO 的使用指南。

GRUB 是 GNU 计划的主要 bootloader。GRUB 最初是由 Erich Boleyn 为 GNU Mach 操作系统撰写的引导程序。后来有 Gordon Matzigkeit 和 Okuji Yoshinori 接替 Erich 的工作，继续维护和开发 GRUB。GRUB 的网站 http://www.gnu.org/software/grub/上有对套件使用的说明文件，叫作《GRUB manual》。GRUB 能够使用 TFTP 和 BOOTP 或者 DHCP 通过网络启动，这种功能对于系统开发过程很有用。

除了传统的 Linux 系统上的引导程序以外，还有其他一些引导程序，也可以支持磁盘引导启动。例如：LoadLin 可以从 DOS 下启动 Linux；还有 ROLO、LinuxBIOS，U-Boot 也支持这种功能。

3．Flash 启动方式

大多数嵌入式系统上都使用 Flash 存储介质。Flash 有很多类型，包括 NOR Flash、NAND

lash 和其他半导体盘。其中，NOR Flash（也就是线性 Flash）使用最为普遍。

NOR Flash 可以支持随机访问，所以代码是可以直接在 Flash 上执行的。Bootloader 一般是存储在 Flash 芯片上的。另外，Linux 内核映像和 RAMDISK 也可以存储在 Flash 上。通常需要把 Flash 分区使用，每个区的大小应该是 Flash 擦除块大小的整数倍。图 6.2 是 Bootloader 和内核映像以及文件系统的分区表。

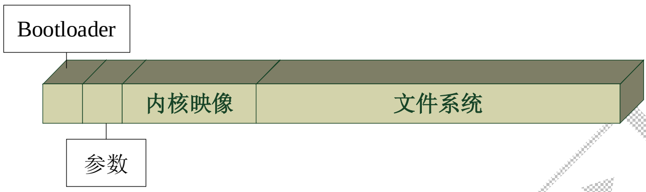


图 6.2　Flash 存储示意图

Bootloader 一般放在 Flash 的底端或者顶端，这要根据处理器的复位向量设置。要使 Bootloader 的入口位于处理器上电执行第一条指令的位置。

接下来分配参数区，这里可以作为 Bootloader 的参数保存区域。

再下来内核映像区。Bootloader 引导 Linux 内核，就是要从这个地方把内核映像解压到 RAM 中去，然后跳转到内核映像入口执行。

然后是文件系统区。如果使用 Ramdisk 文件系统，则需要 Bootloader 把它解压到 RAM 中。如果使用 JFFS2 文件系统，将直接挂接为根文件系统。这两种文件系统将在第 12 章详细讲解。

最后还可以分出一些数据区，这要根据实际需要和 Flash 大小来考虑了。

这些分区是开发者定义的，Bootloader 一般直接读写对应的偏移地址。到了 Linux 内核空间，可以配置成 MTD 设备来访问 Flash 分区。但是，有的 Bootloader 也支持分区的功能，例如：Redboot 可以创建 Flash 分区表，并且内核 MTD 驱动可以解析出 redboot 的分区表。

除了 NOR Flash，还有 NAND Flash、Compact Flash、DiskOnChip 等。这些 Flash 具有芯片价格低，存储容量大的特点。但是这些芯片一般通过专用控制器的 I/O 方式来访问，不能随机访问，因此引导方式跟 NOR Flash 也不同。在这些芯片上，需要配置专用的引导程序。通常，这种引导程序起始的一段代码就把整个引导程序复制到 RAM 中运行，从而实现自举启动，这跟从磁盘上启动有些相似。

6.1.3　Bootloader 的种类

嵌入式系统世界已经有各种各样的 Bootloader，种类划分也有多种方式。除了按照处理器体系结构不同划分以外，还有功能复杂程度的不同。

首先区分一下“Bootloader”和“Monitor”的概念。严格来说，“Bootloader”只是引导设备并且执行主程序的固件；而“Monitor”还提供了更多的命令行接口，可以进行调试、读写内存、烧写 Flash、配置环境变量等。“Monitor”在嵌入式系统开发过程中可以提供很好的调试功能，开发完成以后，就完全设置成了一个“Bootloader”。所以，习惯上大家把它们统称为 Bootloader。

表 6.1 列出了 Linux 的开放源码引导程序及其支持的体系结构。表中给出了 X86 ARM PowerPC 体系结构的常用引导程序，并且注明了每一种引导程序是不是“Monitor”。

表 6.1　　　　开放源码的 Linux 引导程序

Bootloader	Monitor	描　述	x86	ARM	PowerPC
LILO	否	Linux 磁盘引导程序	是	否	否
GRUB	否	GNU 的 LILO 替代程序	是	否	否
Loadlin	否	从 DOS 引导 Linux	是	否	否
ROLO	否	从 ROM 引导 Linux 而不需要 BIOS	是	否	否
Etherboot	否	通过以太网卡启动 Linux 系统的固件	是	否	否
LinuxBIOS	否	完全替代 BUIS 的 Linux 引导程序	是	否	否
BLOB	否	LART 等硬件平台的引导程序	否	是	否
U-boot	是	通用引导程序	是	是	是
RedBoot	是	基于 eCos 的引导程序	是	是	是

对于每种体系结构，都有一系列开放源码 Bootloader 可以选用。

（1）X86

X86 的工作站和服务器上一般使用 LILO 和 GRUB。LILO 是 Linux 发行版主流的 Bootloader。不过 Redhat Linux 发行版已经使用了 GRUB，GRUB 比 LILO 有更有好的显示界面，使用配置也更加灵活方便。

在某些 X86 嵌入式单板机或者特殊设备上，会采用其他 Bootloader，例如：ROLO。这些 Bootloader 可以取代 BIOS 的功能，能够从 FLASH 中直接引导 Linux 启动。现在 ROLO 支持的开发板已经并入 U-Boot，所以 U-Boot 也可以支持 X86 平台。

（2）ARM

ARM 处理器的芯片商很多，所以每种芯片的开发板都有自己的 Bootloader。结果 ARM Bootloader 也变得多种多样。最早有为 ARM720 处理器的开发板的固件，又有了 armboot，StrongARM 平台的 blob，还有 S3C2410 处理器开发板上的 vivi 等。现在 armboot 已经并入了 U-Boot，所以 U-Boot 也支持 ARM/XSCALE 平台。U-Boot 已经成为 ARM 平台事实上的标准 Bootloader。

（3）PowerPC

PowerPC 平台的处理器有标准的 Bootloader，就是 ppcboot。PPCBOOT 在合并 armboot 等之后，创建了 U-Boot，成为各种体系结构开发板的通用引导程序。U-Boot 仍然是 PowerPC 平台的主要 Bootloader。

（4）MIPS

MIPS 公司开发的 YAMON 是标准的 Bootloader，也有许多 MIPS 芯片商为自己的开发板写了 Bootloader。现在，U-Boot 也已经支持 MIPS 平台。

（5）SH

SH 平台的标准 Bootloader 是 sh-boot。Redboot 在这种平台上也很好用。

（6）M68K

M68K 平台没有标准的 Bootloader。Redboot 能够支持 m68k 系列的系统。

值得说明的是 Redboot，它几乎能够支持所有的体系结构，包括 MIPS、SH、M68K 等体系结构。Redboot 是以 eCos 为基础，采用 GPL 许可的开源软件工程。现在由 core eCos 的开

发人员维护，源码下载网站是 http://www.ecoscentric.com/snapshots。Redboot 的文档也相当完善，有详细的使用手册《RedBoot User’s Guide》。

6.2 U-Boot 编程

U-Boot 作为通用的 Bootloader，U-Boot 可以方便地移植到其他硬件平台上，其源代码也值得开发者们研究学习。

6.2.1 U-Boot 工程简介

最早，DENX 软件工程中心的 Wolfgang Denk 基于 8xxrom 的源码创建了 PPCBOOT 工程，并且不断添加处理器的支持。后来，Sysgo Gmbh 把 ppcboot 移植到 ARM 平台上，创建了 ARMboot 工程。然后以 ppcboot 工程和 armboot 工程为基础，创建了 U-Boot 工程。

现在 U-Boot 已经能够支持 PowerPC、ARM、X86、MIPS 体系结构的上百种开发板，已经成为功能最多、灵活性最强并且开发最积极的开放源码 Bootloader。目前仍然由 DENX 的 Wolfgang Denk 维护。

U-Boot 的源码包可以从 sourceforge 网站下载，还可以订阅该网站活跃的 U-Boot Users 邮件论坛，这个邮件论坛对于 U-Boot 的开发和使用都很有帮助。

U-Boot 软件包下载网站：http://sourceforge.net/project/u-boot。

U-Boot 邮件列表网站：http://lists.sourceforge.net/lists/listinfo/u-boot-users/。

DENX 相关的网站：http://www.denx.de/re/DPLG.html。

6.2.2 U-Boot 源码结构

从网站上下载得到 U-Boot 源码包，例如：U-Boot-1.1.2.tar.bz2

解压就可以得到全部 U-Boot 源程序。在顶层目录下有 18 个子目录，分别存放和管理不同的源程序。这些目录中所要存放的文件有其规则，可以分为 3 类。

- 第 1 类目录与处理器体系结构或者开发板硬件直接相关；
- 第 2 类目录是一些通用的函数或者驱动程序；
- 第 3 类目录是 U-Boot 的应用程序、工具或者文档。

表 6.2 列出了 U-Boot 顶层目录下各级目录存放原则。

表 6.2 U-Boot 的源码顶层目录说明

目　录	特　性	解释说明
board	平台依赖	存放电路板相关的目录文件，例如：RPXlite(mpc8xx)、smdk2410(arm920t)、sc520_cdp(x86) 等目录
cpu	平台依赖	存放 CPU 相关的目录文件，例如：mpc8xx、ppc4xx、arm720t、arm920t、xscale、i386 等目录
lib_ppc	平台依赖	存放对 PowerPC 体系结构通用的文件，主要用于实现 PowerPC 平台通用的函数

续表

目　　录	特　　性	解 释 说 明
lib_arm	平台依赖	存放对 ARM 体系结构通用的文件，主要用于实现 ARM 平台通用的函数
lib_i386	平台依赖	存放对 X86 体系结构通用的文件，主要用于实现 X86 平台通用的函数
include	通用	头文件和开发板配置文件，所有开发板的配置文件都在 configs 目录下
common	通用	通用的多功能函数实现
lib_generic	通用	通用库函数的实现
Net	通用	存放网络的程序
Fs	通用	存放文件系统的程序
Post	通用	存放上电自检程序
drivers	通用	通用的设备驱动程序，主要有以太网接口的驱动
Disk	通用	硬盘接口程序
Rtc	通用	RTC 的驱动程序
Dtt	通用	数字温度测量器或者传感器的驱动
examples	应用例程	一些独立运行的应用程序的例子，例如 helloworld
tools	工具	存放制作 S-Record 或者 U-Boot 格式的映像等工具，例如 mkimage
Doc	文档	开发使用文档

U-Boot 的源代码包含对几十种处理器、数百种开发板的支持。可是对于特定的开发板，配置编译过程只需要其中部分程序。这里具体以 S3C2410 arm920t 处理器为例，具体分析 S3C2410 处理器和开发板所依赖的程序，以及 U-Boot 的通用函数和工具。

6.2.3　U-Boot 的编译

U-Boot 的源码是通过 GCC 和 Makefile 组织编译的。顶层目录下的 Makefile 首先可以设置开发板的定义，然后递归地调用各级子目录下的 Makefile，最后把编译过的程序链接成 U-Boot 映像。

1．顶层目录下的 Makefile

它负责 U-Boot 整体配置编译。按照配置的顺序阅读其中关键的几行。

每一种开发板在 Makefile 都需要有板子配置的定义。例如 smdk2410 开发板的定义如下。

```
smdk2410_config :   unconfig
    @./mkconfig $(@:_config=) arm arm920t smdk2410 NULL s3c24x0
```

执行配置 U-Boot 的命令 make smdk2410_config，通过./mkconfig 脚本生成 include/config.mk 的配置文件。文件内容正是根据 Makefile 对开发板的配置生成的。

```
ARCH   = arm
CPU    = arm920t
BOARD  = smdk2410
SOC    = s3c24x0
```

上面的 include/config.mk 文件定义了 ARCH、CPU、BOARD、SOC 这些变量。这样硬件平台依赖的目录文件可以根据这些定义来确定。SMDK2410 平台相关目录如下。

board/smdk2410/
cpu/arm920t/
cpu/arm920t/s3c24x0/
lib_arm/
include/asm-arm/
include/configs/smdk2410.h

再回到顶层目录的 Makefile 文件开始的部分，其中下列几行包含了这些变量的定义。

```
# load ARCH, BOARD, and CPU configuration
include include/config.mk
export      ARCH CPU BOARD VENDOR SOC
```

Makefile 的编译选项和规则在顶层目录的 config.mk 文件中定义。各种体系结构通用的规则直接在这个文件中定义。通过 ARCH、CPU、BOARD、SOC 等变量为不同硬件平台定义不同选项。不同体系结构的规则分别包含在 ppc_config.mk、arm_config.mk、mips_config.mk 等文件中。

顶层目录的 Makefile 中还要定义交叉编译器，以及编译 U-Boot 所依赖的目标文件。

```
ifeq ($(ARCH),arm)
CROSS_COMPILE = arm-linux-         //交叉编译器的前缀
#endif
export  CROSS_COMPILE
…
# U-Boot objects....order is important (i.e. start must be first)
OBJS  = cpu/$(CPU)/start.o                //处理器相关的目标文件
…
LIBS  = lib_generic/libgeneric.a          //定义依赖的目录，每个目录下先把目标
文件连接成*.a 文件。
LIBS += board/$(BOARDDIR)/lib$(BOARD).a
LIBS += cpu/$(CPU)/lib$(CPU).a
ifdef SOC
LIBS += cpu/$(CPU)/$(SOC)/lib$(SOC).a
endif
LIBS += lib_$(ARCH)/lib$(ARCH).a
```

```
…
```

然后还有 U-Boot 映像编译的依赖关系。

```
ALL = u-boot.srec u-boot.bin System.map
all:        $(ALL)
u-boot.srec:    u-boot
            $(OBJCOPY) ${OBJCFLAGS} -O srec $< $@
u-boot.bin: u-boot
            $(OBJCOPY) ${OBJCFLAGS} -O binary $< $@
……
u-boot:         depend $(SUBDIRS) $(OBJS) $(LIBS) $(LDSCRIPT)
            UNDEF_SYM='$(OBJDUMP) -x $(LIBS) \
            |sed  -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'|sort|uniq`;\
            $(LD) $(LDFLAGS) $$UNDEF_SYM $(OBJS) \
                 --start-group $(LIBS) $(PLATFORM_LIBS) --end-group \
                 -Map u-boot.map -o u-boot
```

Makefile 缺省的编译目标为 all，包括 u-boot.srec、u-boot.bin、System.map。u-boot.srec 和 u-boot.bin 又依赖于 U-Boot。U-Boot 就是通过 ld 命令按照 u-boot.map 地址表把目标文件组装成 u-boot。

其他 Makefile 内容就不再详细分析了，上述代码分析应该可以为阅读代码提供了一个线索。

2．开发板配置头文件

除了编译过程 Makefile 以外，还要在程序中为开发板定义配置选项或者参数。这个头文件是 include/configs/<*board_name*>.h。<*board_name*>用相应的 BOARD 定义代替。

这个头文件中主要定义了两类变量。

一类是选项，前缀是 CONFIG_，用来选择处理器、设备接口、命令、属性等。例如：

```
#define   CONFIG_ARM920T       1
#define   CONFIG_DRIVER_CS8900  1
```

另一类是参数，前缀是 CFG_，用来定义总线频率、串口波特率、Flash 地址等参数。例如：

```
#define     CFG_FLASH_BASE      0x00000000
#define CFG_PROMPT              "=>"
```

3．编译结果

根据对 Makefile 的分析，编译分为 2 步。第 1 步配置，例如：make smdk2410_config；第 2 步编译，执行 make 就可以了。

编译完成后，可以得到 U-Boot 各种格式的映像文件和符号表，如表 6.3 所示。

表 6.3　　U-Boot 编译生成的映像文件

文 件 名 称	说　　明	文 件 名 称	说　　明
System.map	U-Boot 映像的符号表	u-boot.bin	U-Boot 映像原始的二进制格式
u-boot	U-Boot 映像的 ELF 格式	u-boot.srec	U-Boot 映像的 S-Record 格式

U-Boot 的 3 种映像格式都可以烧写到 Flash 中，但需要看加载器能否识别这些格式。一般 u-boot.bin 最为常用，直接按照二进制格式下载，并且按照绝对地址烧写到 Flash 中就可以了。U-Boot 和 u-boot.srec 格式映像都自带定位信息。

4．U-Boot 工具

在 tools 目录下还有些 U-Boot 的工具。这些工具有的也经常用到。表 6.4 说明了几种工具的用途。

表 6.4　　U-Boot 的工具

工 具 名 称	说　　明	工 具 名 称	说　　明
bmp_logo	制作标记的位图结构体	img2srec	转换 SREC 格式映像
envcrc	校验 u-boot 内部嵌入的环境变量	mkimage	转换 U-Boot 格式映像
gen_eth_addr	生成以太网接口 MAC 地址	updater	U-Boot 自动更新升级工具

这些工具都有源代码，可以参考改写其他工具。其中 mkimage 是很常用的一个工具，Linux 内核映像和 ramdisk 文件系统映像都可以转换成 U-Boot 的格式。

6.2.4　U-Boot 的移植

U-Boot 能够支持多种体系结构的处理器，支持的开发板也越来越多。因为 Bootloader 是完全依赖硬件平台的，所以在新电路板上需要移植 U-Boot 程序。

开始移植 U-Boot 之前，先要熟悉硬件电路板和处理器。确认 U-Boot 是否已经支持新开发板的处理器和 I/O 设备。假如 U-Boot 已经支持一块非常相似的电路板，那么移植的过程将非常简单。

移植 U-Boot 工作就是添加开发板硬件相关的文件、配置选项，然后配置编译。

开始移植之前，需要先分析一下 U-Boot 已经支持的开发板，比较出硬件配置最接近的开发板。选择的原则是，首先处理器相同，其次处理器体系结构相同，然后是以太网接口等外围接口。还要验证一下这个参考开发板的 U-Boot，至少能够配置编译通过。

以 S3C2410 处理器的开发板为例，U-Boot-1.1.2 版本已经支持 SMDK2410 开发板。我们可以基于 SMDK2410 移植，那么先把 SMDK2410 编译通过。

我们以 S3C2410 开发板 fs2410 为例说明。移植的过程参考 SMDK2410 开发板，SMDK2410 在 U-Boot-1.1.2 中已经支持。

移植 U-Boot 的基本步骤如下。

（1）在顶层 Makefile 中为开发板添加新的配置选项，使用已有的配置项目为例。

```
smdk2410_config  :     unconfig
```

```
        @./mkconfig $(@:_config=) arm arm920t smdk2410 NULL s3c24x0
```

参考上面 2 行，添加下面 2 行。

```
fs2410_config   :       unconfig
      @./mkconfig $(@:_config=) arm arm920t fs2410 NULL s3c24x0
```

（2）创建一个新目录存放开发板相关的代码，并且添加文件。

board/fs2410/config.mk

board/fs2410/flash.c

board/fs2410/fs2410.c

board/fs2410/Makefile

board/fs2410/memsetup.S

board/fs2410/u-boot.lds

（3）为开发板添加新的配置文件

可以先复制参考开发板的配置文件，再修改。例如：

$cp include/configs/smdk2410.h   include/configs/fs2410.h

如果是为一颗新的 CPU 移植，还要创建一个新的目录存放 CPU 相关的代码。

（4）配置开发板

$ make fs2410_config

（5）编译 U-Boot

执行 make 命令，编译成功可以得到 U-Boot 映像。有些错误是跟配置选项是有关系的，通常打开某些功能选项会带来一些错误，一开始可以尽量跟参考板配置相同。

（6）添加驱动或者功能选项

在能够编译通过的基础上，还要实现 U-Boot 的以太网接口、Flash 擦写等功能。

对于 FS2410 开发板的以太网驱动和 smdk2410 完全相同，所以可以直接使用。CS8900 驱动程序文件如下。

drivers/cs8900.c

drivers/cs8900.h

对于 Flash 的选择就麻烦多了，Flash 芯片价格或者采购方面的因素都有影响。多数开发板大小、型号不都相同。所以还需要移植 Flash 的驱动。每种开发板目录下一般都有 flash.c 这个文件，需要根据具体的 Flash 类型修改。例如：

board/fs2410/flash.c

（7）调试 U-Boot 源代码，直到 U-Boot 在开发板上能够正常启动。

调试的过程可能是很艰难的，需要借助工具，并且有些问题可能困扰很长时间。

6.2.5 添加 U-Boot 命令

U-Boot 的命令为用户提供了交互功能，并且已经实现了几十个常用的命令。如果开发板需要很特殊的操作，可以添加新的 U-Boot 命令。

U-Boot 的每一个命令都是通过 U_Boot_CMD 宏定义的。这个宏在 include/command.h 头

文件中定义，每一个命令定义一个 cmd_tbl_t 结构体。

```
#define U_BOOT_CMD(name,maxargs,rep,cmd,usage,help) \
cmd_tbl_t __u_boot_cmd_##name Struct_Section = {#name, maxargs, rep, cmd,
usage, help}
```

这样每一个 U-Boot 命令有一个结构体来描述。结构体包含的成员变量：命令名称、最大参数个数、重复数、命令执行函数、用法、帮助。

从控制台输入的命令是由 common/command.c 中的程序解释执行的。find_cmd()负责匹配输入的命令，从列表中找出对应的命令结构体。

基于 U-Boot 命令的基本框架，来分析一下简单的 icache 操作命令，就可以知道添加新命令的方法。

（1）定义 CACHE 命令。在 include/cmd_confdefs.h 中定义了所有 U-Boot 命令的标志位。

```
#define CFG_CMD_CACHE      0x00000010ULL   /* icache, dcache      */
```

如果有更多的命令，也要在这里添加定义。

（2）实现 CACHE 命令的操作函数。下面是 common/cmd_cache.c 文件中 icache 命令部分的代码。

```
#if (CONFIG_COMMANDS & CFG_CMD_CACHE)
static int on_off (const char *s)
{       //这个函数解析参数，判断是打开 cache，还是关闭 cache
       if (strcmp(s, "on") == 0) { //参数为“on”
              return (1);
       } else if (strcmp(s, "off") == 0) { //参数为“off”
              return (0);
    }
    return (-1);
}
int do_icache ( cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])
{     //对指令 cache 的操作函数
      switch (argc) {
      case 2:               /* 参数个数为 1，则执行打开或者关闭指令 cache 操作 */
            switch (on_off(argv[1])) {
            case 0:    icache_disable();       //打开指令 cache
                  break;
            case 1:    icache_enable ();       //关闭指令 cache
                  break;
            }
            /* FALL TROUGH */
```

```
        case 1:           /* 参数个数为 0，则获取指令 cache 状态*/
              printf ("Instruction Cache is %s\n",
                     icache_status() ? "ON" : "OFF");
              return 0;
        default:  //其他缺省情况下，打印命令使用说明
              printf ("Usage:\n%s\n", cmdtp->usage);
              return 1;
        }
        return 0;
}
……
U_Boot_CMD( //通过宏定义命令
   icache,   2,   1,    do_icache, //命令为 icache，命令执行函数为 do_icache()
   "icache  - enable or disable instruction cache\n", //帮助信息
   "[on, off]\n"
   "    - enable or disable instruction cache\n"
);
……
#endif
```

U-Boot 的命令都是通过结构体__U_Boot_cmd_##name 来描述的。根据 U_Boot_CMD 在 include/command.h 中的两行定义可以明白。

```
#define U_BOOT_CMD(name,maxargs,rep,cmd,usage,help) \
cmd_tbl_t __u_boot_cmd_##name Struct_Section = {#name, maxargs, rep, cmd, usage, help}
```

还有，不要忘了在 common/Makefile 中添加编译的目标文件。

（3）打开 CONFIG_COMMANDS 选项的命令标志位。这个程序文件开头有#if 语句需要预处理是否包含这个命令函数。CONFIG_COMMANDS 选项在开发板的配置文件中定义。例如：SMDK2410 平台在 include/configs/smdk2410.h 中有如下定义。

```
/***********************************************************
 * Command definition
 ***********************************************************/
#define CONFIG_COMMANDS \
              (CONFIG_CMD_DFL   | \
              CFG_CMD_CACHE     | \
              CFG_CMD_REGINFO    | \
              CFG_CMD_DATE     | \
              CFG_CMD_ELF)
```

按照这 3 步，就可以添加新的 U-Boot 命令。

;.3 U-Boot 的调试

新移植的 U-Boot 不能正常工作，这时就需要调试了。调试 U-Boot 离不开工具，只有理
军 U-Boot 启动过程，才能正确地调试 U-Boot 源码。

6.3.1 硬件调试器

硬件电路板制作完成以后，这时上面还没有任何程序，就叫作裸板。首要的工作是把程
予或者固件加载到裸板上，这就要通过硬件工具来完成。习惯上，这种硬件工具叫作仿真器。

仿真器可以通过处理器的 JTAG 等接口控制板子，直接把程序下载到目标板内存，或者进行
lash 编程。如果板上的 Flash 是可以拔插的，就可以通过专用的 Flash 烧写器来完成。在第 4 章
个绍过目标板跟主机之间的连接，其中 JTAG 等接口就是专门用来连接仿真器的。

仿真器还有一个重要的功能就是在线调试程序，这对于调试 Bootloader 和硬件测试程序
艮有用。

从最简单的 JTAG 电缆，到 ICE 仿真器，再到可以调试 Linux 内核的仿真器。

复杂的仿真器可以支持与计算机间的以太网或者 USB 接口通信。

对于 U-Boot 的调试，可以采用 BDI2000。BDI2000 完全可以反汇编地跟踪 Flash 中的程
予，也可以进行源码级的调试。

使用 BDI2000 调试 U-boot 的方法如下。

（1）配置 BDI2000 和目标板初始化程序，连接目标板。

（2）添加 U-Boot 的调试编译选项，重新编译。

U-Boot 的程序代码是位置相关的，调试的时候尽量在内存中调试，可以修改连接定位地
上 TEXT_BASE。TEXT_BASE 在 board/<board_name>/config.mk 中定义。

另外，如果有复位向量也需要先从链接脚本中去掉。链接脚本是 board/<board_name>/
-boot.lds。

添加调试选项，在 config.mk 文件中查找，DBGFLAGS，加上-g 选项。然后重新编译 U-Boot。

（3）下载 U-Boot 到目标板内存。

通过 BDI2000 的下载命令 LOAD，把程序加载到目标板内存中。然后跳转到 U-Boot 入口。

（4）启动 GDB 调试。

启动 GDB 调试，这里是交叉调试的 GDB。GDB 与 BDI2000 建立链接，然后就可以设
置断点执行了。

```
$ arm-linux-gdb u-boot
(gdb)target remote 192.168.1.100:2001
(gdb)stepi
(gdb)b start_armboot
(gdb)c
```

6.3.2 软件跟踪

假如 U-Boot 没有任何串口打印信息，手头又没有硬件调试工具，那样怎么知道 U-Boot

执行到什么地方了呢？可以通过开发板上的 LED 指示灯判断。

开发板上最好设计安装八段数码管等 LED，可以用来显示数字或者数字位。

U-Boot 可以定义函数 show_boot_progress (int status)，用来指示当前启动进度。在 include/common.h 头文件中声明这个函数。

```
#ifdef CONFIG_SHOW_BOOT_PROGRESS
void    show_boot_progress (int status);
#endif
```

CONFIG_SHOW_BOOT_PROGRESS 是需要定义的。这个在板子配置的头文件中定义。CSB226 开发板对这项功能有完整实现，可以参考。在头文件 include/configs/csb226.h 中，有下列一行。

```
#define CONFIG_SHOW_BOOT_PROGRESS      1
```

函数 show_boot_progress (int status)的实现跟开发板关系密切，所以一般在 board 目录下的文件中实现。看一下 CSB226 在 board/csb226/csb226.c 中的实现函数。

```
/** 设置CSB226板的 0、1、2 三个指示灯的开关状态
 * csb226_set_led: - switch LEDs on or off
 * @param led:   LED to switch (0,1,2)
 * @param state: switch on (1) or off (0)
 */
void csb226_set_led(int led, int state)
{
     switch(led) {
           case 0: if (state==1) {
                         GPCR0 |= CSB226_USER_LED0;
                 } else if (state==0) {
                        GPSR0 |= CSB226_USER_LED0;
                 }
                 break;
           case 1: if (state==1) {
                         GPCR0 |= CSB226_USER_LED1;
                 } else if (state==0) {
                         GPSR0 |= CSB226_USER_LED1;
                 }
                 break;
           case 2: if (state==1) {
                         GPCR0 |= CSB226_USER_LED2;
                } else if (state==0) {
```

```
                        GPSR0 |= CSB226_USER_LED2;
                    }
                    break;
            }
            return;
    }
    /** 显示启动进度函数，在比较重要的阶段，设置三个灯为亮的状态（1, 5, 15）*/
    void show_boot_progress (int status)
    {
            switch(status) {
                    case  1: csb226_set_led(0,1); break;
                    case  5: csb226_set_led(1,1); break;
                    case 15: csb226_set_led(2,1); break;
            }
            return;
    }
```

这样，在 U-Boot 启动过程中就可以通过 show_boot_progresss 指示执行进度。比如 hang() 函数是系统出错时调用的函数，这里需要根据特定的开发板给定显示的参数值。

```
void hang (void)
{
        puts ("### ERROR ### Please RESET the board ###\n");
#ifdef CONFIG_SHOW_BOOT_PROGRESS
        show_boot_progress(-30);
#endif
        for (;;);
}
```

6.3.3 U-Boot 启动过程

尽管有了调试跟踪手段，甚至也可以通过串口打印信息了，但是不一定能够判断出错原因。如果能够充分理解代码的启动流程，那么对准确地解决和分析问题很有帮助。

开发板上电后，执行 U-Boot 的第一条指令，然后顺序执行 U-Boot 启动函数。函数调用顺序如图 6.3 所示。

看一下 board/smsk2410/u-boot.lds 这个链接脚本，可以知道目标程序的各部分链接顺序。第一个要链接的是 cpu/arm920t/start.o，那么 U-Boot 的入口指令一定位于这个程序中。下面详细分析一下程序跳转和函数的调用关系以及函数实现。

1．cpu/arm920t/start.S

这个汇编程序是 U-Boot 的入口程序，开头就是复位向量的代码。

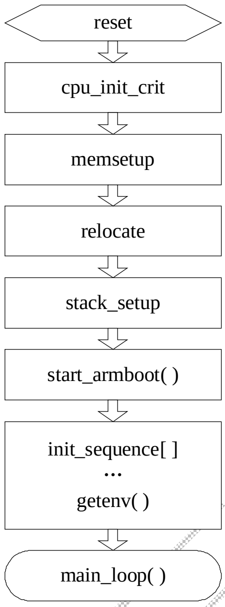


图 6.3 U-Boot 启动代码流程图

```
_start: b      reset        //复位向量
      ldr   pc, _undefined_instruction
      ldr   pc, _software_interrupt
      ldr   pc, _prefetch_abort
      ldr   pc, _data_abort
      ldr   pc, _not_used
      ldr   pc, _irq      //中断向量
      ldr   pc, _fiq      //中断向量
…
 /* the actual reset code  */
reset:          //复位启动子程序
      /* 设置 CPU 为 SVC32 模式 */
      mrs   r0,cpsr
      bic   r0,r0,#0x1f
      orr   r0,r0,#0xd3
      msr   cpsr,r0
/* 关闭看门狗 */

/* 这些初始化代码在系统重起的时候执行，运行时热复位从 RAM 中启动不执行 */
#ifdef CONFIG_INIT_CRITICAL
      bl    cpu_init_crit
#endif
```

```
relocate:                    /* 把 U-Boot 重新定位到 RAM*/
      adr   r0, _start         /* r0 是代码的当前位置 */
      ldr   r1, _TEXT_BASE     /* 测试判断是从 Flash 启动，还是 RAM */
      cmp    r0, r1        /* 比较 r0 和 r1，调试的时候不要执行重定位 */
      beq    stack_setup   /* 如果 r0 等于 r1，跳过重定位代码 */
      /* 准备重新定位代码 */
      ldr   r2, _armboot_start
      ldr   r3, _bss_start
      sub   r2, r3, r2         /* r2 得到 armboot 的大小   */
      add   r2, r0, r2         /* r2 得到要复制代码的末尾地址 */
copy_loop: /* 重新定位代码 */
      ldmia r0!, {r3-r10}   /*从源地址[r0]复制 */
      stmia r1!, {r3-r10}   /* 复制到目的地址[r1] */
      cmp   r0, r2          /* 复制数据块直到源数据末尾地址[r2] */
      ble   copy_loop

      /* 初始化堆栈等    */
stack_setup:
      ldr   r0, _TEXT_BASE             /* 上面是 128 KiB 重定位的 u-boot */
      sub   r0, r0, #CFG_MALLOC_LEN    /* 向下是内存分配空间 */
      sub   r0, r0, #CFG_GBL_DATA_SIZE /* 然后是 bdinfo 结构体地址空间  */
#ifdef CONFIG_USE_IRQ
      sub   r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)
#endif
      sub   sp, r0, #12     /* 为 abort-stack 预留 3 个字 */
clear_bss:
      ldr   r0, _bss_start     /* 找到 bss 段起始地址 */
      ldr   r1, _bss_end       /*  bss 段末尾地址   */
      mov   r2, #0x00000000    /* 清零 */
clbss_l:str r2, [r0]        /* bss 段地址空间清零循环...  */
      add   r0, r0, #4
      cmp   r0, r1
      bne   clbss_l
      /* 跳转到 start_armboot 函数入口，_start_armboot 字保存函数入口指针 */
      ldr   pc, _start_armboot
_start_armboot: .word start_armboot     //start_armboot 函数在 lib_arm/board.c
中实现
/* 关键的初始化子程序 */
```

```
cpu_init_crit:
……    //初始化CACHE，关闭MMU等操作指令
        /* 初始化RAM时钟。
        * 因为内存时钟是依赖开发板硬件的，所以在board的相应目录下可以找到memsetup.S文件。
        */
        mov   ip, lr
        bl    memsetup        //memsetup子程序在board/smdk2410/memsetup.S中实现
        mov   lr, ip
        mov   pc, lr
```

2．lib_arm/board.c

start_armboot是U-Boot执行的第一个C语言函数，完成系统初始化工作，进入主循环，处理用户输入的命令。

```
void start_armboot (void)
{
        DECLARE_GLOBAL_DATA_PTR;
        ulong size;
        init_fnc_t **init_fnc_ptr;
        char *s;
        /* Pointer is writable since we allocated a register for it */
        gd = (gd_t*)(_armboot_start - CFG_MALLOC_LEN - sizeof(gd_t));
        /* compiler optimization barrier needed for GCC >= 3.4 */
        __asm__ __volatile__("": : :"memory");
        memset ((void*)gd, 0, sizeof (gd_t));
        gd->bd = (bd_t*)((char*)gd - sizeof(bd_t));
        memset (gd->bd, 0, sizeof (bd_t));
        monitor_flash_len = _bss_start - _armboot_start;
        /* 顺序执行init_sequence数组中的初始化函数 */
        for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
              if ((*init_fnc_ptr)() != 0) {
                     hang ();
              }
        }
        /*配置可用的Flash */
        size = flash_init ();
        display_flash_config (size);
        /* _armboot_start 在u-boot.lds链接脚本中定义 */
        mem_malloc_init (_armboot_start - CFG_MALLOC_LEN);
```

```
        /* 配置环境变量，重新定位 */
        env_relocate ();
        /* 从环境变量中获取 IP 地址 */
        gd->bd->bi_ip_addr = getenv_IPaddr ("ipaddr");
        /* 以太网接口 MAC 地址 */
        ……
        devices_init ();      /* 获取列表中的设备 */
        jumptable_init ();
        console_init_r ();    /* 完整地初始化控制台设备 */
        enable_interrupts ();/* 使能例外处理 */
        /* 通过环境变量初始化 */
        if ((s = getenv ("loadaddr")) != NULL) {
               load_addr = simple_strtoul (s, NULL, 16);
        }
        /* main_loop()总是试图自动启动，循环不断执行 */
        for (;;) {
               main_loop ();      /* 主循环函数处理执行用户命令 -- common/main.c */
        }
        /* NOTREACHED - no way out of command loop except booting */
}
```

3．init_sequence[]

init_sequence[]数组保存着基本的初始化函数指针。这些函数名称和实现的程序文件在下列注释中。

```
init_fnc_t *init_sequence[] = {
        cpu_init,             /* 基本的处理器相关配置 -- cpu/arm920t/cpu.c */
        board_init,           /* 基本的板级相关配置 -- board/smdk2410/smdk2410.c */
        interrupt_init,       /* 初始化例外处理 -- cpu/arm920t/s3c24x0/interrupt.c */
        env_init,             /* 初始化环境变量 -- common/cmd_flash.c */
        init_baudrate,        /* 初始化波特率设置 -- lib_arm/board.c */
        serial_init,          /* 串口通讯设置 -- cpu/arm920t/s3c24x0/serial.c */
        console_init_f,       /* 控制台初始化阶段 1 -- common/console.c */
        display_banner,       /* 打印 u-boot 信息 -- lib_arm/board.c */
        dram_init,            /* 配置可用的 RAM -- board/smdk2410/smdk2410.c */
        display_dram_config,  /* 显示 RAM 的配置大小 -- lib_arm/board.c */
        NULL,
};
```

6.3.4　U-Boot 与内核的关系

U-Boot 作为 Bootloader，具备多种引导内核启动的方式。常用的 go 和 bootm 命令可以直接引导内核映像启动。U-Boot 与内核的关系主要是内核启动过程中参数的传递。

1．go 命令的实现

```
/* common/cmd_boot.c  */
int do_go (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])
{
      ulong addr, rc;
      int    rcode = 0;
      if (argc < 2) {
            printf ("Usage:\n%s\n", cmdtp->usage);
            return 1;
      }
      addr = simple_strtoul(argv[1], NULL, 16);
      printf ("## Starting application at 0x%08lX ...\n", addr);
      /*
       * pass address parameter as argv[0] (aka command name),
       * and all remaining args
       */
      rc = ((ulong (*)(int, char *[]))addr) (--argc, &argv[1]);
      if (rc != 0) rcode = 1;

      printf ("## Application terminated, rc = 0x%lX\n", rc);
      return rcode;
}
```

go 命令调用 do_go()函数，跳转到某个地址执行的。如果在这个地址准备好了自引导的内核映像，就可以启动了。尽管 go 命令可以带变参，实际使用时一般不用来传递参数。

2．bootm 命令的实现

```
/* common/cmd_bootm.c */
int do_bootm (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])
{
      ulong iflag;
      ulong addr;
      ulong data, len, checksum;
      ulong  *len_ptr;
```

```
        uint  unc_len = 0x400000;
        int   i, verify;
        char  *name, *s;
        int   (*appl)(int, char *[]);
        image_header_t *hdr = &header;

        s = getenv ("verify");
        verify = (s && (*s == 'n')) ? 0 : 1;
        if (argc < 2) {
              addr = load_addr;
        } else {
              addr = simple_strtoul(argv[1], NULL, 16);
        }
        SHOW_BOOT_PROGRESS (1);
        printf ("## Booting image at %08lx ...\n", addr);
        /* Copy header so we can blank CRC field for re-calculation */
        memmove (&header, (char *)addr, sizeof(image_header_t));
        if (ntohl(hdr->ih_magic) != IH_MAGIC)
        {
              puts ("Bad Magic Number\n");
              SHOW_BOOT_PROGRESS (-1);
              return 1;
        }
        SHOW_BOOT_PROGRESS (2);
        data = (ulong)&header;
        len  = sizeof(image_header_t);

        checksum = ntohl(hdr->ih_hcrc);
        hdr->ih_hcrc = 0;

        if(crc32 (0, (char *)data, len) != checksum) {
              puts ("Bad Header Checksum\n");
              SHOW_BOOT_PROGRESS (-2);
              return 1;
        }
        SHOW_BOOT_PROGRESS (3);
        /* for multi-file images we need the data part, too */
        print_image_hdr ((image_header_t *)addr);
        data = addr + sizeof(image_header_t);
```

```
        len  = ntohl(hdr->ih_size);
        if(verify) {
              puts ("   Verifying Checksum ... ");
              if(crc32 (0, (char *)data, len) != ntohl(hdr->ih_dcrc)) {
                    printf ("Bad Data CRC\n");
                    SHOW_BOOT_PROGRESS (-3);
                    return 1;
              }
              puts ("OK\n");
        }
        SHOW_BOOT_PROGRESS (4);
        len_ptr = (ulong *)data;
……
        switch (hdr->ih_os) {
        default:                /* handled by (original) Linux case */
        case IH_OS_LINUX:
             do_bootm_linux  (cmdtp, flag, argc, argv,
                        addr, len_ptr, verify);
             break;
        ……
}
```

bootm 命令调用 do_bootm 函数。这个函数专门用来引导各种操作系统映像，可以支持引导 Linux、vxWorks、QNX 等操作系统。引导 Linux 的时候，调用 do_bootm_linux()函数。

3．do_bootm_linux 函数的实现

```
/* lib_arm/armlinux.c */
void do_bootm_linux (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[],
                ulong addr, ulong *len_ptr, int verify)
{
        DECLARE_GLOBAL_DATA_PTR;
        ulong len = 0, checksum;
        ulong initrd_start, initrd_end;
        ulong data;
        void (*theKernel)(int zero, int arch, uint params);
        image_header_t *hdr = &header;
        bd_t *bd = gd->bd;
#ifdef CONFIG_CMDLINE_TAG
        char *commandline = getenv ("bootargs");
```

```
#endif
      theKernel = (void (*)(int, int, uint))ntohl(hdr->ih_ep);
      /* Check if there is an initrd image */
      if(argc >= 3) {
            SHOW_BOOT_PROGRESS (9);
            addr = simple_strtoul (argv[2], NULL, 16);
            printf ("## Loading Ramdisk Image at %08lx ...\n", addr);
            /* Copy header so we can blank CRC field for re-calculation */
            memcpy (&header, (char *) addr, sizeof (image_header_t));
            if (ntohl (hdr->ih_magic) != IH_MAGIC) {
                   printf ("Bad Magic Number\n");
                   SHOW_BOOT_PROGRESS (-10);
                   do_reset (cmdtp, flag, argc, argv);
            }
            data = (ulong) & header;
            len = sizeof (image_header_t);
            checksum = ntohl (hdr->ih_hcrc);
            hdr->ih_hcrc = 0;
            if(crc32 (0, (char *) data, len) != checksum) {
                  printf ("Bad Header Checksum\n");
                  SHOW_BOOT_PROGRESS (-11);
                  do_reset (cmdtp, flag, argc, argv);
            }
            SHOW_BOOT_PROGRESS (10);
            print_image_hdr (hdr);
            data = addr + sizeof (image_header_t);
            len = ntohl (hdr->ih_size);
            if(verify) {
                  ulong csum = 0;
                  printf ("   Verifying Checksum ... ");
                  csum = crc32 (0, (char *) data, len);
                  if (csum != ntohl (hdr->ih_dcrc)) {
                        printf ("Bad Data CRC\n");
                        SHOW_BOOT_PROGRESS (-12);
                        do_reset (cmdtp, flag, argc, argv);
                  }
                  printf ("OK\n");
            }
            SHOW_BOOT_PROGRESS (11);
```

```
            if ((hdr->ih_os != IH_OS_LINUX) ||
                  (hdr->ih_arch != IH_CPU_ARM) ||
                  (hdr->ih_type != IH_TYPE_RAMDISK)) {
                  printf ("No Linux ARM Ramdisk Image\n");
                  SHOW_BOOT_PROGRESS (-13);
                  do_reset (cmdtp, flag, argc, argv);
            }
            /* Now check if we have a multifile image */
      } else if ((hdr->ih_type == IH_TYPE_MULTI) && (len_ptr[1])) {
             ulong tail = ntohl (len_ptr[0]) % 4;
             int i;
             SHOW_BOOT_PROGRESS (13);
             /* skip kernel length and terminator */
             data = (ulong) (&len_ptr[2]);
             /* skip any additional image length fields */
             for (i = 1; len_ptr[i]; ++i)
                    data += 4;
            /* add kernel length, and align */
            data += ntohl (len_ptr[0]);
            if (tail) {
                    data += 4 - tail;
            }
            len = ntohl (len_ptr[1]);
      } else {
             /* no initrd image */
            SHOW_BOOT_PROGRESS (14);
            len = data = 0;
      }
      if (data) {
             initrd_start = data;
             initrd_end = initrd_start + len;
      } else {
             initrd_start = 0;
             initrd_end = 0;
      }
      SHOW_BOOT_PROGRESS (15);
      debug ("## Transferring control to Linux (at address %08lx) ...\n",
             (ulong) theKernel);
#if defined (CONFIG_SETUP_MEMORY_TAGS) || \
```

```
      defined (CONFIG_CMDLINE_TAG) || \
      defined (CONFIG_INITRD_TAG) || \
      defined (CONFIG_SERIAL_TAG) || \
      defined (CONFIG_REVISION_TAG) || \
      defined (CONFIG_LCD) || \
      defined (CONFIG_VFD)
      setup_start_tag (bd);
#ifdef CONFIG_SERIAL_TAG
      setup_serial_tag (&params);
#endif
#ifdef CONFIG_REVISION_TAG
      setup_revision_tag (&params);
#endif
#ifdef CONFIG_SETUP_MEMORY_TAGS
      setup_memory_tags (bd);
#endif
#ifdef CONFIG_CMDLINE_TAG
      setup_commandline_tag (bd, commandline);
#endif
#ifdef CONFIG_INITRD_TAG
      if (initrd_start && initrd_end)
              setup_initrd_tag (bd, initrd_start, initrd_end);
#endif
      setup_end_tag (bd);
#endif
      /* we assume that the kernel is in place */
      printf ("\nStarting kernel ...\n\n");
      cleanup_before_linux ();

      theKernel (0, bd->bi_arch_number, bd->bi_boot_params);
}
```

do_bootm_linux()函数是专门引导 Linux 映像的函数，它还可以处理 ramdisk 文件系统的映像。这里引导的内核映像和 ramdisk 映像，必须是 U-Boot 格式的。U-Boot 格式的映像可以通过 mkimage 工具来转换，其中包含了 U-Boot 可以识别的符号。

6.4　使用 U-Boot

U-Boot 是“Monitor”。除了 Bootloader 的系统引导功能，它还有用户命令接口，提供了

一些复杂的调试、读写内存、烧写 Flash、配置环境变量等功能。掌握 U-Boot 的使用，将极大地方便嵌入式系统的开发。

6.4.1 烧写 U-Boot 到 Flash

新开发的电路板没有任何程序可以执行，也就不能启动，需要先将 U-Boot 烧写到 Flash 中。

如果主板上的 EPROM 或者 Flash 能够取下来，就可以通过编程器烧写。例如：计算机 BIOS 就存储在一块 256KB 的 Flash 上，通过插座与主板连接。

但是多数嵌入式单板使用贴片的 Flash，不能取下来烧写。这种情况可以通过处理器的调试接口，直接对板上的 Flash 编程。

处理器调试接口是为处理器芯片设计的标准调试接口，包含 BDM、JTAG 和 EJTAG 3 种接口标准。JTAG 接口在第 4 章已经介绍过；BDM（Background Debug Mode）主要应用在 PowerPC8xx 系列处理器上；EJTAG 主要应用在 MIPS 处理器上。这 3 种硬件接口标准定义有所不同，但是功能基本相同，下面都统称为 JTAG 接口。

JTAG 接口需要专用的硬件工具来连接。无论从功能、性能角度，还是从价格角度，这些工具都有很大差异。关于这些工具的选择，将在第 6.4.1 节详细介绍。

最简单方式就是通过 JTAG 电缆，转接到计算机并口连接。这需要在主机端开发烧写程序，还需要有并口设备驱动程序。开发板上电或者复位的时候，烧写程序探测到处理器并且开始通信，然后把 Bootloader 下载并烧写到 Flash 中。这种方式速率很慢，可是价格非常便宜。一般来说，平均每秒钟可以烧写 100～200 个字节。

烧写完成后，复位实验板，串口终端应该显示 U-Boot 的启动信息。

6.4.2 U-Boot 的常用命令

U-Boot 上电启动后，敲任意键可以退出自动启动状态，进入命令行。

```
U-Boot 1.1.2 (Apr 26 2005 - 12:27:13)
U-Boot code: 11080000 -> 1109614C  BSS: -> 1109A91C
RAM Configuration:
Bank #0: 10000000 32 MB
Micron StrataFlash MT28F128J3 device initialized
Flash: 32 MB
In:    serial
Out:   serial
Err:   serial
Hit any key to stop autoboot:  0
U-Boot>
```

在命令行提示符下，可以输入 U-Boot 的命令并执行。U-Boot 可以支持几十个常用命令，通过这些命令，可以对开发板进行调试，可以引导 Linux 内核，还可以擦写 Flash 完成系统部署等功能。掌握这些命令的使用，才能够顺利地进行嵌入式系统的开发。

输入 help 命令，可以得到当前 U-Boot 的所有命令列表。每一条命令后面是简单的命令说明。

```
=> help
?       - alias for 'help'
autoscr - run script from memory
base    - print or set address offset
bdinfo  - print Board Info structure
boot    - boot default, i.e., run 'bootcmd'
bootd   - boot default, i.e., run 'bootcmd'
bootm   - boot application image from memory
bootp   - boot image via network using BootP/TFTP protocol
cmp     - memory compare
coninfo  - print console devices and information
cp      - memory copy
crc32   - checksum calculation
dhcp    - invoke DHCP client to obtain IP/boot params
echo    - echo args to console
erase   - erase FLASH memory
flinfo  - print FLASH memory information
go      - start application at address 'addr'
help    - print online help
iminfo  - print header information for application image
imls    - list all images found in flash
itest    - return true/false on integer compare
loadb   - load binary file over serial line (kermit mode)
loads   - load S-Record file over serial line
loop   - infinite loop on address range
md    - memory display
mm    - memory modify (auto-incrementing)
mtest   - simple RAM test
mw      - memory write (fill)
nfs     - boot image via network using NFS protocol
nm      - memory modify (constant address)
printenv - print environment variables
protect - enable or disable FLASH write protection
rarpboot - boot image via network using RARP/TFTP protocol
reset   - Perform RESET of the CPU
run     - run commands in an environment variable
```

```
saveenv - save environment variables to persistent storage
setenv  - set environment variables
sleep   - delay execution for some time
tftpboot - boot image via network using TFTP protocol
version - print monitor version
=>
```

U-Boot 还提供了更加详细的命令帮助，通过 help 命令还可以查看每个命令的参数说明。由于开发过程的需要，有必要先把 U-Boot 命令的用法弄清楚。接下来，根据每一条命令的帮助信息，解释一下这些命令的功能和参数。

```
=> help bootm
bootm [addr [arg ...]]
    - boot application image stored in memory
          passing arguments 'arg ...'; when booting a Linux kernel,
          'arg' can be the address of an initrd image
```

bootm 命令可以引导启动存储在内存中的程序映像。这些内存包括 RAM 和可以永久保存的 Flash。

第 1 个参数 addr 是程序映像的地址，这个程序映像必须转换成 U-Boot 的格式。

第 2 个参数对于引导 Linux 内核有用，通常作为 U-Boot 格式的 RAMDISK 映像存储地址；也可以是传递给 Linux 内核的参数（缺省情况下传递 bootargs 环境变量给内核）。

```
=> help bootp
bootp [loadAddress] [bootfilename]
```

bootp 命令通过 bootp 请求，要求 DHCP 服务器分配 IP 地址，然后通过 TFTP 协议下载指定的文件到内存。

第 1 个参数是下载文件存放的内存地址。

第 2 个参数是要下载的文件名称，这个文件应该在开发主机上准备好。

```
=> help cmp
cmp [.b, .w, .l] addr1 addr2 count
     - compare memory
```

cmp 命令可以比较 2 块内存中的内容。.b 以字节为单位；.w 以字为单位；.l 以长字为单位。注意：cmp.b 中间不能保留空格，需要连续敲入命令。

第 1 个参数 addr1 是第一块内存的起始地址。

第 2 个参数 addr2 是第二块内存的起始地址。

第 3 个参数 count 是要比较的数目，单位按照字节、字或者长字。

```
=> help cp
cp [.b, .w, .l] source target count
       - copy memory
```

cp 命令可以在内存中复制数据块，包括对 Flash 的读写操作。

第 1 个参数 source 是要复制的数据块起始地址。

第 2 个参数 target 是数据块要复制到的地址。这个地址如果在 Flash 中，那么会直接调用写 Flash 的函数操作。所以 U-Boot 写 Flash 就使用这个命令，当然需要先把对应 Flash 区域擦干净。

第 3 个参数 count 是要复制的数目，根据 cp.b cp.w cp.l 分别以字节、字、长字为单位。

```
=> help crc32
crc32 address count [addr]
    - compute CRC32 checksum [save at addr]
```

crc32 命令可以计算存储数据的校验和。

第 1 个参数 address 是需要校验的数据起始地址。

第 2 个参数 count 是要校验的数据字节数。

第 3 个参数 addr 用来指定保存结果的地址。

```
=> help echo
echo [args..]
     - echo args to console; \c suppresses newline
```

echo 命令回显参数。

```
=> help erase
erase start end
     - erase FLASH from addr 'start' to addr 'end'
erase N:SF[-SL]
     - erase sectors SF-SL in FLASH bank # N
erase bank N
     - erase FLASH bank # N
erase all
     - erase all FLASH banks
```

erase 命令可以擦 Flash。

参数必须指定 Flash 擦除的范围。

按照起始地址和结束地址，start 必须是擦除块的起始地址；end 必须是擦除末尾块的结束地址。这种方式最常用。举例说明：擦除 0x20000 – 0x3ffff 区域命令为 erase 20000 3ffff。

按照组和扇区，N 表示 Flash 的组号，SF 表示擦除起始扇区号，SL 表示擦除结束扇区号。另外，还可以擦除整个组，擦除组号为 N 的整个 Flash 组。擦除全部 Flash 只要给出一个 all 的参数即可。

```
=> help flinfo
flinfo
      - print information for all FLASH memory banks
```

```
flinfo N
      - print information for FLASH memory bank # N
```

flinfo 命令打印全部 Flash 组的信息，也可以只打印其中某个组。一般嵌入式系统的 Flash 只有一个组。

```
=> help go
go addr [arg ...]
     - start application at address 'addr'
       passing 'arg' as arguments
```

go 命令可以执行应用程序。
第 1 个参数是要执行程序的入口地址。
第 2 个可选参数是传递给程序的参数，可以不用。

```
=> help iminfo
iminfo addr [addr ...]
     - print header information for application image starting at
       address 'addr' in memory; this includes verification of the
       image contents (magic number, header and payload checksums)
```

iminfo 可以打印程序映像的开头信息，包含了映像内容的校验（序列号、头和校验和）。
第 1 个参数指定映像的起始地址。
可选的参数是指定更多的映像地址。

```
=> help loadb
loadb [ off ] [ baud ]
    - load binary file over serial line with offset 'off' and baudrate 'baud'
```

loadb 命令可以通过串口线下载二进制格式文件。

```
=> help loads
loads [ off ]
   - load S-Record file over serial line with offset 'off'
```

loads 命令可以通过串口线下载 S-Record 格式文件。

```
=> help mw
mw [.b, .w, .l] address value [count]
    - write memory
```

mw 命令可以按照字节、字、长字写内存，.b .w .l 的用法与 cp 命令相同。
第 1 个参数 address 是要写的内存地址。
第 2 个参数 value 是要写的值。
第 3 个可选参数 count 是要写单位值的数目。

```
=> help nfs
nfs [loadAddress] [host ip addr:bootfilename]
```

nfs 命令可以使用 NFS 网络协议通过网络启动映像。

```
=> help nm
nm [.b, .w, .l] address
    - memory modify, read and keep address
```

nm 命令可以修改内存，可以按照字节、字、长字操作。
参数 address 是要读出并且修改的内存地址。

```
=> help printenv
printenv
     - print values of all environment variables
printenv name ...
     - print value of environment variable 'name'
```

printenv 命令打印环境变量。
可以打印全部环境变量，也可以只打印参数中列出的环境变量。

```
=> help protect
protect on  start end
     - protect Flash from addr 'start' to addr 'end'
protect on  N:SF[-SL]
     - protect sectors SF-SL in Flash bank # N
protect on  bank N
     - protect Flash bank # N
protect on  all
     - protect all Flash banks
protect off start end
     - make Flash from addr 'start' to addr 'end' writable
protect off N:SF[-SL]
     - make sectors SF-SL writable in Flash bank # N
protect off bank N
     - make Flash bank # N writable
protect off all
     - make all Flash banks writable
```

protect 命令是对 Flash 写保护的操作，可以使能和解除写保护。
第 1 个参数 on 代表使能写保护；off 代表解除写保护。
第 2、3 参数是指定 Flash 写保护操作范围，跟擦除的方式相同。

```
=> help rarpboot
rarpboot [loadAddress] [bootfilename]
```

rarboot 命令可以使用 TFTP 协议通过网络启动映像。也就是把指定的文件下载到指定地址，然后执行。

第 1 个参数是映像文件下载到的内存地址。

第 2 个参数是要下载执行的映像文件。

```
=> help run
run var [...]
     - run the commands in the environment variable(s) 'var'
```

run 命令可以执行环境变量中的命令，后面参数可以跟几个环境变量名。

```
=> help setenv
setenv name value ...
     - set environment variable 'name' to 'value ...'
setenv name
     - delete environment variable 'name'
```

setenv 命令可以设置环境变量。

第 1 个参数是环境变量的名称。

第 2 个参数是要设置的值，如果没有第 2 个参数，表示删除这个环境变量。

```
=> help sleep
sleep N
     - delay execution for N seconds (N is _decimal_ !!!)
```

sleep 命令可以延迟 N 秒钟执行，N 为十进制数。

```
=> help tftpboot
tftpboot [loadAddress] [bootfilename]
```

tftpboot 命令可以使用 TFTP 协议通过网络下载文件。按照二进制文件格式下载。另外使用这个命令，必须配置好相关的环境变量。例如 serverip 和 ipaddr。

第 1 个参数 loadAddress 是下载到的内存地址。

第 2 个参数是要下载的文件名称，必须放在 TFTP 服务器相应的目录下。

这些 U-Boot 命令为嵌入式系统提供了丰富的开发和调试功能。在 Linux 内核启动和调试过程中，都可以用到 U-Boot 的命令。但是一般情况下，不需要使用全部命令。比如已经支持以太网接口，可以通过 tftpboot 命令来下载文件，那么还有必要使用串口下载的 loadb 吗？反过来，如果开发板需要特殊的调试功能，也可以添加新的命令。

在建立交叉开发环境和调试 Linux 内核等章节时，在 ARM 平台上移植了 U-Boot，并且提供了具体 U-Boot 的操作步骤。

6.4.3　U-Boot 的环境变量

有点类似 Shell，U-Boot 也使用环境变量。可以通过 printenv 命令查看环境变量的设置。

```
U-Boot> printenv
bootdelay=3
baudrate=115200
netmask=255.255.0.0
ethaddr=12:34:56:78:90:ab
bootfile=uImage
bootargs=console=ttyS0,115200 root=/dev/ram rw initrd=0x30800000,8M
bootcmd=tftp 0x30008000 zImage;go 0x30008000
serverip=192.168.1.1
ipaddr=192.168.1.100
stdin=serial
stdout=serial
stderr=serial

Environment size: 337/131068 bytes
U-Boot>
```

表 6.5 是常用环境变量的含义解释。通过 printenv 命令可以打印出这些变量的值。

表 6.5　　U-Boot 环境变量的解释说明

环 境 变 量	解 释 说 明
bootdelay	定义执行自动启动的等候秒数
baudrate	定义串口控制台的波特率
netmask	定义以太网接口的掩码
ethaddr	定义以太网接口的 MAC 地址
bootfile	定义缺省的下载文件
bootargs	定义传递给 Linux 内核的命令行参数
bootcmd	定义自动启动时执行的几条命令
serverip	定义 tftp 服务器端的 IP 地址
ipaddr	定义本地的 IP 地址
stdin	定义标准输入设备，一般是串口
stdout	定义标准输出设备，一般是串口
stderr	定义标准出错信息输出设备，一般是串口

U-Boot 的环境变量都可以有缺省值，也可以修改并且保存在参数区。U-Boot 的参数区一般有 EEPROM 和 Flash 两种设备。

环境变量的设置命令为 setenv，在 6.2.2 节有命令的解释。

举例说明环境变量的使用。

```
=>setenv serverip 192.168.1.1
=>setenv ipaddr 192.168.1.100
=>setenv rootpath "/usr/local/arm/3.3.2/rootfs"
=>setenv bootargs "root=/dev/nfs rw nfsroot=\$(serverip):\$(rootpath) ip=
$(ipaddr) "
=>setenv kernel_addr 30000000
=>setenv nfscmd "tftp \$(kernel_addr) uImage; bootm \$(kernel_addr) "
=>run nfscmd
```

上面定义的环境变量有 serverip ipaddr rootpath bootargs kernel_addr。环境变量 bootargs 中还使用了环境变量，bootargs 定义命令行参数，通过 bootm 命令传递给内核。环境变量 nfscmd 中也使用了环境变量，功能是把 uImage 下载到指定的地址并且引导起来。可以通过 run 命令执行 nfscmd 脚本。

“黑色经典”系列之《嵌入式 Linux 系统开发技术详解——基于 ARM》

第 7 章 配置编译内核

本章目标

本章介绍了 Linux 2.6 内核的特点和配置编译。通过学习本章，可以了解 Linux 2.6 内核的 kbuild 编译管理方式，掌握基本的配置编译过程。

Linux 内核特点 □
配置编译内核源码 □

7.1 Linux 内核特点

7.1.1 Linux 内核版本介绍

Linux 内核的版本号分为主版本号、次版本号和扩展版本号等。根据稳定版本、测试版本和开发版本定义不同版本序列。

稳定版本的主版本号用偶数表示，例如：2.2、2.4、2.6。每隔 2～3 年启动一个 Linux 稳定主版本号。

紧接着是次版本号，例如：2.6.13、2.6.14、2.6.15。次版本号不分奇偶数，顺序递增。每隔 1～2 个月发布一个稳定版本。

然后是升级版本号，例如：2.6.14.3、2.6.14.4、2.6.14.5。升级版本号不分奇偶数，顺序递增。每周几次发布升级版本号，修正最新的稳定版本的问题。

另外一种是测试版本。在下一个稳定版本发布之前，每个月发布几个测试版本，例如：2.6.12-rc1。通过测试，可以使内核正式发布的时候更加稳定。

还有一类是开发版本。开发版本的主版本号用奇数表示，例如：2.3、2.5。也有次版本号，例如：2.5.32、2.5.33。开发版本是不稳定的，适合内核开发者在新的稳定的主版本发布之前使用。

7.1.2 Linux 内核特点

（1）Linux 内核的重要特点可移植性（Portability），支持硬件平台广泛，在大多数体系结构上都可以运行。

可量测性（Scalability），即可以运行在超级计算机上，也可以运行在很小的设备上（4MB RAM 就能满足）。

标准化和互用性（Interoperability），遵守标准化和互用性规范。

完善的网络支持。

安全性，开放源码使缺陷暴露无疑，它的代码也接受了许多专家的审查。

稳定性（Stability）和可靠性（Reliability）。

模块化（Modularity），运行时可以根据系统的需要加载程序。

编程容易，可以学习现有的代码，还可以从网络上找到很多有用的资源。

（2）Linux 内核支持的处理器体系结构

Linux 内核能够支持的处理器的最小要求：32 位处理器，带或者不带 MMU。需要说明的是，不带 MMU 的处理器过去是 uClinux 支持的。Linux 2.6 内核采纳了 m68k 等不带 MMU 的部分平台，Linux 支持的绝大多数处理器还是带 MMU 的。

Linux 内核既能支持 32 位体系结构，又能支持 64 位体系结构。

每一种体系结构在内核源码树的 arch/目录下有子目录。各种体系结构的详细内容可以查看源码 Documentation/<arch>/目录下的文档。

（3）Linux 内核遵守的软件许可

Linux 内核全部源代码是遵守 GPL 软件许可的免费软件，这就要求在发布 Linux 软件的时候免费开放源码。

对于 Linux 等自由软件，必须对最终用户开放源代码，但是没有义务向其他任何人开放。在商业 Linux 公司中，通常会要求客户签署最终用户的使用许可。

私有的模块是允许使用的。只要不被认定为源自 GPL 的代码，就可以按照私有许可使用。但是，私有的驱动程序不能静态链接到内核中去，可以作为动态加载的模块使用。

（4）开放源码驱动程序的优点

基于庞大的 Linux 社区和内核源码工程，有各种各样的驱动程序和应用程序可以利用，而没有必要从头写程序。

开发者可以免费得到社区的贡献、支持、检查代码和测试。驱动程序可以免费发布给其他人，可以静态编译进内核。

对 Linux 公司来说，用户和社区的正面印象可以使他们更容易聘请到有才能的开发者。

以源码形式发布驱动程序，可以不必为每一个内核版本和补丁版本都提供二进制的程序。另外通过分析源代码，可以保证它没有安全隐患。

7.1.3 Linux 2.6 内核新特性

Linux 2.6 内核吸收了一些新技术，在性能、可量测性、支持和可用性方面不断提高。这些改进多数是添加支持更多的体系结构、处理器、总线、接口和设备；也有一些是标准化内部接口，简化扩展添加新设备和子系统的支持。

与 Linux 2.4 版本相比，Linux 2.6 版本具有许多新特性，内核也有很大修改。其中一些修改只跟内核或者驱动开发者有关，另外一些修改则会影响到系统启动、系统管理和应用程序开发。

Linux 2.6 内核重要的新特性如下。

（1）新的调度器

Linux 2.6 版本的 Linux 内核使用了新的调度器算法，它是由 Ingo Molnar 开发的 O(1)调度器算法。它在高负载的情况下极其出色，并且对多处理器调度有很好的扩展。

Linux 2.4 版本的标准调度器中，使用时间片重算的算法。这种算法要求在所有的进程都用尽时间片以后，重新计算下一次运行的时间片。这样每次任务调度的花销不确定，可能因为计算比较复杂，产生较大调度延迟。特别是多处理器系统，可能由于调度的延迟，导致大部分处理器处于空闲状态，影响系统性能。

新的调度器采用 O(1)的调度算法，通过优先级数组的数据结构来实现。优先级数组可以使每个优先级都有相应的任务队列，还有一个优先级位图。每个优先级对应位图中一位，通过位图可以快速执行最高优先级任务。因为优先级个数是固定的，所以查找的时间也固定，不受系统运行任务数的影响。

新的调度器为每个处理器维护 2 个优先级数组：有效数组和过期数组。有效数组内任务队列的进程都还有可以运行的时间片；过期数组内任务队列的进程都已经没有时间片可以执行。当一个进程的时间片用光时，就把它从有效数组移到过期数组，并且时间片也已经重新计算好了。当需要重新调度这些任务的时候，只要在有效数组和过期数组之间切换就好了。这种交换是 O(1)算法的核心。它根本不需要从头到尾重新计算所有任务的时间片，调度器的效率更高。

O(1)调度器具有以下优点。

- SMP 效率高。如果有工作需要完成，那么所有处理器都会工作。
- 没有进程需要长时间地等待处理器；也没有进程会无端地占用大量的 CPU 时间。
- SMP 进程只映射到一个 CPU 而且不会在 CPU 之间跳跃。
- 不重要的任务可以设置低优先级，重要的任务设置高优先级。
- 负载平衡功能。调度器会降低那些超出处理器负载能力的进程的优先级。
- 交互性能提高。即使在高负载的情况下，也不会再发生长时间不响应鼠标点击或者键盘输入的情况。

（2）内核抢占

Linux 2.6 采纳了内核抢占的补丁，大大减小了用户交互、多媒体等应用程序的调度延迟。这一特性对实时系统和嵌入式系统来说特别有用。这项工作是由 Robert Love 完成的。

在 Linux 2.4 以前的内核版本中，内核空间运行的任务（包括通过系统调用进入内核空间的用户任务）不允许被抢占，直到它们自己主动释放 CPU。

在 Linux 2.6 内核中，内核是可抢占的。一个内核任务可以被抢占，为的是让重要的用户应用程序可以继续运行。这样做可以极大地增强系统的用户交互性，用户将会觉得鼠标点击和击键的事件得到了更快速的响应。

当然，不是所有的内核代码段都可以被抢占。可以锁定内核代码的关键部分，不允许抢占。这样可以确保每个 CPU 的数据结构和状态始终受到保护。

（3）新的线程模型

Linux 2.6 内核重写了线程框架。它也是由 Ingo Molnar 完成的。它基于一个 1:1 的线程模型，能够支持 NPTL（Native Posix Threading Library）线程库。NPTL 是一个改进的 Linux 线程库，它是由 Molnar 和 Ulrich Drepper 合作开发的。

对于 2.4 内核的 Linux 线程库，存在一些不足。例如：总是需要一个管理线程，来负责创建和删除子线程，负责接收和分发信号等。如果系统中使用大量的线程，这种 Linux 线程库就存在严重的效率问题。

NPTL 线程库解决了传统的 Linux 线程库存在的问题，对系统有很大性能提升。实际上，RedHat 已经将它向后移植到了 Linux 2.4 内核，从 RedHat 9.0 版本就开始包含对它的支持。新的线程框架的改进包含 Linux 线程空间中的许多新的概念，包括线程组、线程各自的本地存储区、POSIX 风格的信号以及其他改进。

（4）文件系统

相对于 Linux 2.4，Linux 2.6 对文件系统的支持在很多方面都有大的改进。关键的变化包括对扩展属性（extended attributes）以及 POSIX 标准的访问控制（access controls）的支持。

EXT2/EXT3 文件系统作为多数 Linux 系统缺省安装的文件系统，是在 2.6 中改进最大的一个。最主要的变化是对扩展属性的支持，也即给指定的文件在文件系统中嵌入一些元数据（metadata）。新的扩展属性子系统的第一个用途就是实现 POSIX 访问控制链表。POSIX 访问控制是标准 UNIX 权限控制的超集，支持更细粒度的访问控制。EXT3 还有其他一些细微变化。

Linux 对文件系统层还进行了大量的改进以兼容其他操作系统。Linux 2.6 对 NTFS 文件系统的支持也进行了重写；同时也支持 IBM 的 JFS(journaling file system)和 SGI 的 XFS。

此外，Linux 文件系统中还有很多零散的变化。

（5）声音

Linux 2.6 内核还添加了新的声音系统：ALSA(Advanced Linux Sound Architecture)。老的声音系统 OSS (Open Sound System)存在一些系统结构的缺陷。新的声音体系结构支持 USB 音频和 MIDI 设备，全双工重放等。

（6）总线

Linux 2.6 的 IDE/ATA、SCSI 等存储总线也都被更新。最主要的是重写了 IDE 子系统，解决了许多可扩展性问题以及其他限制。其次是可以象微软的 Windows 操作系统那样检测介质的变动，以更好地兼容那些并不完全遵照标准规范的设备。

Linux 2.6 还大大提升了对 PCI 总线的支持，增强或者扩展了 USB、蓝牙（Bluetooth）、红外（IrDA）等外围设备总线。所有的总线设备类型（硬件、无线和存储）都集成到了 Linux 新的设备模型子系统中。

（7）电源管理

Linux 2.6 支持高级电源配置管理界面（ACPI，Advanced Configuration and Power Interface），最早 Linux 2.4 中有些支持。ACPI 不同于 APM（高级电源管理），拥有这种接口的系统在改变电源状态时需要分别通知每一个兼容的设备。新的内核系统允许子系统跟踪需要进行电源状态转换的设备。

（8）网络

Linux 是一种网络性能优越的操作系统，已经可以支持世界上大多数主流网络协议，包括 TCP/IP（IPv4/IPv6）、AppleTalk、IPX 等。

在网络硬件驱动方面，利用了 Linux 的设备模型底层的改进和许多设备驱动程序的升级。例如，Linux 2.6 提供一个独立的 MII（媒体独立接口，或是 IEEE 802.3u）子系统，它被许多网络设备驱动程序使用。新的子系统替换了原先系统中各自运行的多个实例，消除了原先系统中多个驱动程序使用重复代码、采用类似的方法处理设备的 MII 支持的情况。

在网络安全方面，Linux 2.6 的一个重要改进是提供了对 IPsec 协议的支持。IPsec 是在网络协议层为 IPv4 和 IPv6 提供加密支持的一组协议。由于安全是在协议层提供的，对应用层是透明的。它与 SSL 协议及其他 tunneling/security 协议很相似，但是位于一个低得多的层面。当前内核支持的加密算法包括 SHA（“安全散列算法”）、DES（“数据加密标准”）等。

在协议方面，Linux 2.6 还加强了对多播网络的支持。网络多播使得由一点发出的数据包可以被多台计算机接收（传统的点对点网络每次只能有两方通信）。

还有其他一些改进。例如：IPv6 已经成熟；VLAN 的支持也已经成熟等。

（9）用户界面层

Linux 2.6 中一个主要的内部改动是人机接口层的大量重写。人机接口层是一个 Linux 系统中用户体验的中心，包括视频输出、鼠标、键盘等。内核的新版本中，这一层的重写以及模块化工作超出了以前的任何一个版本。

Linux 2.6 对显示器输出处理的支持也有不少改进，但大部分只在配置使用内核内部的帧缓冲控制台子系统时才有用。

人机界面层还加入了对近乎所有可接入设备的支持，从触摸屏到盲人用的设备，到各种各样的鼠标。

（10）统一的设备模型

Linux 2.6 内核最值得关注的变化是创建了一个统一的设备模型。这个设备模型通过维持大量的数据结构囊括了几乎所有的设备结构和系统。这样做的好处是，可以改进设备的电源管理和简化设备相关的任务管理。

这种设备模型可以跟踪获取以下信息。

- 系统中存在的设备，其所连接的总线。
- 特定情形下设备的电源状态。
- 系统清楚设备的驱动程序，并清楚哪些设备受其控制。
- 系统的总线结构，哪个设备连接在哪个总线上，以及哪些总线互连（例如，USB 和 PCI 总线的互连）。
- 设备在系统中的类别描述（类别包括磁盘，分区等）。

Linux 2.6 内核引入了 sysfs 文件系统，提供了系统的设备模型的用户空间描述。通常 sysfs 文件系统挂接在/sys 目录下。

7.2　配置编译内核源码

在广大爱好者的支持下，Linux 内核版本不断更新。新的内核修订了旧内核的 bug，并增加了许多新的特性。如果用户想要使用这些新特性，或想根据自己的系统度身定制一个更高效、更稳定的内核，就需要重新编译内核。

通常，新的内核会支持更多的硬件，具备更好的进程管理能力，运行速度更快、更稳定，并且一般会修复老版本中发现的许多漏洞等，经常性地选择升级更新的系统内核是 Linux 使用者的必要操作内容。

为了正确、合理地设置内核编译配置选项，从而只编译系统需要的功能的代码，一般主要有下面 4 个考虑。

（1）尺寸小。自己定制内核可以使代码尺寸减小，运行将会更快。

（2）节省内存。由于内核部分代码永远占用物理内存，定制内核可以使系统拥有更多的可用物理内存。

（3）减少漏洞。不需要的功能编译进入内核可能会增加被系统攻击者利用的机会。

（4）动态加载模块。根据需要动态地加载或者卸载模块，可以节省系统内存。但是，将某种功能编译为模块方式会比编译到内核内的方式速度要慢一些。

7.2.1　内核源码结构

由于内核版本是不断升级更新的，最好下载使用最新版本的内核源代码。但是，有时候也需要比较分析老版本的内核。

浏览 http://kernel.org 站点，可以查看 Linux 官方发布的内核版本，从而确定需要的内核版本。然后可以通过 HTTP 或者 FTP 下载相应的源码包。

Linux 的下载工具，例如：gftp、kget、wget 等。其中，wget 下载工具就很好用，它可

以支持 FTP 和 HTTP，还支持断点续传，不过是命令行的方式。下面都以 wget 为例来下载源码包。

下例就是下载内核源码包和电子签名文件到当前目录。由于源码包一般都在 30MB 以上，可以使用断点续传的下载方式，加上选项“-c”。下载命令如下。

```
$ wget  -c http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.14.tar.bz2
$ wget  http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.14.tar.bz2.sign
```

下载完成以后，先验证一下电子签名。

```
$ gpg –verify linux-2.4.26.tar.bz2.sign
```

如果没有问题，就可以使用源码包了。可以把源码包解压到工作目录下。

```
$ cd ~/workspace
$ tar jxvf ~/linux-2.6.14.tar.bz2
```

新版本的内核分两种，一种是完整源码版本，另外一种是 patch 文件，即补丁。完整的内核版本比较大，一般是 tar.gz 或者是.bz2 文件，二者分别是使用 gzip 或者 bzip2 进行压缩的文件，使用时需要解压缩。patch 文件则比较小，一般只有几十 K 到几百 K，但是 patch 文件是针对于特定的版本的，你需要找到自己对应的版本才能使用。

每一个补丁都反映了最近的 2 个正式版本之间的差别。也就是说，上一个版本的 Linux 内核源码，通过打补丁可以得到下一个版本。

另外，Linux 社区经常有开发版本、分支版本或者非官方修改，都是以补丁的形式发布的。

假设已经下载了 Linux-2.6.14 版本，kernel.org 又发布了 Linux-2.6.15 内核。这时下载补丁 patch-2.6.14.15，就可以升级到新的版本。下载命令如下。

```
$ wget http://www.kernel.org/pub/linux/kernel/v2.6/patch-2.6.14.15.bz2
$ wget http://www.kernel.org/pub/linux/kernel/v2.6/patch-2.6.14.15.bz2.sign
```

下载完成后，也要检查电子签名。

```
$ gpg –verify patch-2.6.14.15.bz2.sign
```

通常使用*.bz2 文件，因为*.bz2 的压缩比更高一些。这里我们不需要解压补丁文件，直接使用 bzcat 来读取文件信息就行了。

```
$ cd linux-2.6.14/
$ bzcat ../patch-2.6.14.15.bz2 | patch –p1
```

上面通过管道的方式，把补丁内容传递给 patch 命令，应用到内核源代码中去。然后，可以把 Linux-2.6.14 的目录名称改成 Linux-2.6.15，就得到新版本的 Linux 内核源码了。

那么补丁文件是什么呢？不妨分析一下补丁文件的内容。补丁文件是通过 diff 命令比较两个源码目录中文件的结果，把两个目录中所有文件的变化体现出来。下面是补丁文件中的

一段，说明了 Makefile 文件的一些修改。

```
diff --git a/Makefile b/Makefile
index 1fa7e53..497884d 100644
--- a/Makefile
+++ b/Makefile
@@ -1,8 +1,8 @@
 VERSION = 2
 PATCHLEVEL = 6
-SUBLEVEL = 14
+SUBLEVEL = 15
 EXTRAVERSION =
-NAME=Affluent Albatross
+NAME=Sliding Snow Leopard
```

上面是 a 目录和 b 目录比较的结果，也就是从 a 目录到 b 目录的变化。“–”表示删除当前行，“+”表示添加当前行，这样可以实现代码的修改替换。上面的 SUBLEVEL 从 14 变成了 15。

patch 命令可以根据补丁文件内容修改指定目录下的文件。几种命令使用方式如下。

```
$ patch –p<n> < diff_file
$ cat diff_file | patch –p<n>
$ bzcat diff_file.bz2 | patch –p<n>
$ zcat diff_file.gz | patch –p<n>
```

其中，<n>代表按照 patch 文件的路径忽略的目录级数，每个“/”代表一级。例如：

p0 是完全按照补丁文件中的路径查找要修改的文件；

p1 则使用去掉第一级“/”得到相对路径，再基于当前目录，到相应的相对路径下查找要修改的文件。

接下来，就可以仔细阅读内核源代码。Linux 内核源代码非常庞大，随着版本的发展不断增加。它使用目录树结构，并且使用 Makefile 组织配置编译。

初次接触 Linux 内核，要仔细阅读顶层目录的 readme，它是 Linux 内核的概述和编译命令说明。readme 的说明更加针对 X86 等通用的平台，对于某些特殊的体系结构，可能有些特殊的地方。

顶层目录的 Makefile 是整个内核配置编译的核心文件，负责组织目录树中子目录的编译管理，还可以设置体系结构和版本号等。

内核源码的顶层有许多子目录，分别组织存放各种内核子系统或者文件。具体的目录说明见表 7.1。

表 7.1　　Linux 内核源码顶层目录说明

arch/	体系结构相关的代码，例如：arch/i386，arch/arm，arch/ppc

crypto	
drivers/	各种设备驱动程序，例如：drivers/char drivers/block …
Documentation/	内核文档
fs/	文件系统，例如：fs/ext3/ fs/jffs2 …
include/	内核头文件： include/asm 是体系结构相关的头文件，它是 include/asm-arm、include/asm-i386 等目录的链接。 include/linux 是 Linux 内核基本的头文件
init/	Linux 初始化，例如：main.c
ipc/	进程间通信的代码
kernel/	Linux 内核核心代码（这部分很小）
lib/	各种库子程序，例如：zlib， crc32
mm/	内存管理代码
net/	网络支持代码，主要是网络协议
sound	声音驱动的支持
scripts/	内部或者外部使用的脚本
usr/	用户的代码

7.2.2 内核配置系统

Linux 内核源代码支持二十多种体系结构的处理器，还有各种各样的驱动程序等选项。因此，在编译之前必须根据特定平台配置内核源代码。Linux 内核有上千个配置选项，配置相当复杂。所以，Linux 内核源代码组织了一个配置系统。

Linux 内核配置系统可以生成内核配置菜单，方便内核配置。配置系统主要包含 Makefile、Kconfig 和配置工具，可以生成配置界面。配置界面是通过工具来生成的，工具通过 Makefile 编译执行，选项则是通过各级目录的 Kconfig 文件定义。

Linux 内核配置命令有：make config、make menuconfig 和 make xconfig。分别是字符界面、ncurses 光标菜单和 X-window 图形窗口的配置界面。字符界面配置方式需要回答每一个选项提示，逐个回答内核上千个选项几乎是行不通的；图形窗口的配置界面很好，光标菜单也方便实用。例如执行 make xconfig，主菜单界面如图 7.1 所示。

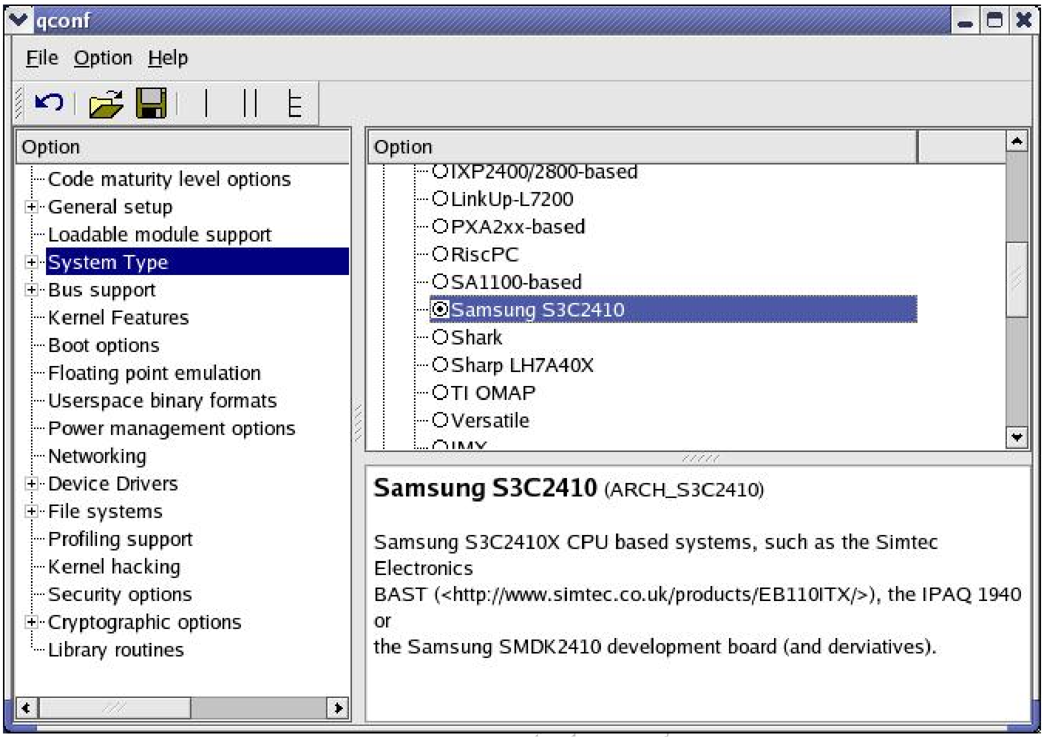


图 7.1 内核图形配置界面

那么这个配置界面到底是如何生成的呢？这里结合配置系统的 3 个部分分析一下。

1．Makefile

Linux 内核的配置编译都是由顶层目录的 Makefile 整体管理的。顶层目录的 Makefile 定义了配置和编译的规则。关于 Makefile 的具体使用方法可以参考第 3 章的内容，这里重点分析相关的变量和规则。

参考内核源码包中的 Documentation/kbuild/makefiles.txt，可以得到内核使用 Makefile 的详细说明。

在顶层的 Makefile 中，可以查找到如下几行定义的规则。

```
config %config: scripts_basic outputmakefile FORCE
    $(Q)mkdir -p include/linux
    $(Q)$(MAKE) $(build)=scripts/kconfig $@
```

这就是生成内核配置界面的命令规则，它也定义了执行的目标和依赖的前提条件，还有要执行的命令。

这条规则定义的目标为 config %config，通配符%意味着可以包括 config、xconfig、gconfig、menuconfig 和 oldconfig 等。依赖的前提条件是 scripts_basic outputmakefile，这些在 Makefile 也是规则定义，主要用来编译生成配置工具。

那么这条规则执行的命令就是执行 scripts/kconfig/Makefile 指定的规则。相当于：

```
make -C scripts/kconfig/  config
```

或者

```
make -C scripts/kconfig/  %config
```

这两行命令是使用配置工具解析 arch/$(ARCH)/Kconfig 文件，生成内核配置菜单。$(ARCH)变量是 Linux 体系结构定义，对应 arch 目录下子目录的名称。Kconfig 包含了内核配置菜单的内容，那么 arch/$(ARCH)/Kconfig 是配置主菜单的文件，调用管理其他各级 Kconfig。

根据配置工具的不同，内核也有不同的配置方式。有命令行方式，还有图形界面方式。表 7.2 是各种内核配置方式的说明。

表 7.2　　内核配置方式说明

配 置 方 式	功　　能
config	通过命令行程序更新当前配置
menuconfig	通过菜单程序更新当前配置
xconfig	通过 QT 图形界面更新当前配置
gconfig	通过 GTK 图形界面更新当前配置
oldconfig	通过已经提供的.config 文件更新当前配置
randconfig	对所有的选项随机配置
defconfig	对所有选项使用缺省配置
allmodconfig	对所有选项尽可能选择“m”
allyesconfig	对所有选项尽可能选择“y”
allnoconfig	对所有选项尽可能选择“n”的最小配置

这些内核配置方式是在 scripts/kconfig/Makefile 中通过规则定义的。从这个 Makefile 中，可以找到下面一些规则定义。如果把变量或者通配符带进去，就可以明白要执行的操作。这里的 ARCH 以 arm 为例来说明。

```
xconfig: $(obj)/qconf
      $< arch/$(ARCH)/Kconfig
```

执行命令：scripts/kconfig/qconf　arch/arm/Kconfig

使用 QT 图形库，生成内核配置界面。arch/arm/Kconfig 是菜单的主配置文件，每种配置方式都需要。

```
gconfig: $(obj)/gconf
      $< arch/$(ARCH)/Kconfig
```

执行命令：scripts/kconfig/gconf　arch/arm/Kconfig

使用 GTK 图形库，生成内核配置界面。

```
menuconfig: $(obj)/mconf
    $(Q)$(MAKE) $(build)=scripts/lxdialog
    $< arch/$(ARCH)/Kconfig
```

执行命令：scripts/kconfig/mconf arch/arm/Kconfig
使用 lxdialog 工具，生成光标配置菜单。
因为 mconf 调用 lxdialog 工具，所以需要先编译 scripts/lxdialog 目录。

```
config: $(obj)/conf
     $< arch/$(ARCH)/Kconfig
```

执行命令：scripts/kconfig/conf arch/arm/Kconfig
完全命令行的内核配置方式。

```
oldconfig: $(obj)/conf
     $< -o arch/$(ARCH)/Kconfig
```

执行命令：scripts/kconfig/conf -o arch/arm/Kconfig
完全命令行的内核配置方式。使用“-o”选项，直接读取已经存在的.config 文件，要求确认内核新的配置项。

```
silentoldconfig: $(obj)/conf
     $< -s arch/$(ARCH)/Kconfig
```

执行命令：scripts/kconfig/conf -s arch/arm/Kconfig
完全命令行的内核配置方式。使用“-s”选项，直接读取已经存在的.config 文件，提示但不要求确认内核新的配置项。

```
%_defconfig: $(obj)/conf
    $(Q)$< -D arch/$(ARCH)/configs/$@ arch/$(ARCH)/Kconfig
```

执行命令：scripts/kconfig/conf -D arch/arm/configs/%_defconfig arch/arm/Kconfig
完全命令行的内核配置方式。读取缺省的配置文件 arch/arm/configs/%_defconfig，另存成.config 文件。

通过上述各种都可以完成配置内核的工作，在顶层目录下生成.config 文件。这个.config 文件保存大量的内核配置项，.config 会自动转换成 include/linux/autoconf.h 头文件。在 include/linux/config.h 文件中，将包含使用 include/linux/autoconf.h 头文件。

2．配置工具

不同的内核配置方式，分别通过不同的配置工具来完成。scripts 目录下提供了各种内核配置工具，表 7.3 是这些工具的说明。

表 7.3　　内核配置工具说明

配　置　工　具	Makefile 相关目标	依赖的程序和软件

conf	defconfig oldconfig …	conf.c zconf.tab.c
mconf	menuconfig	mconf.c zconf.tab.c 调用 scripts/lxdialog/lxdialog
qconf	xconfig	qconf.c kconfig_load.c zconf.tab.c 基于 QT 软件包实现图形界面
gconf	gconfig	gconf.c kconfig_load.c zconf.tab.c 基于 GTK 软件包实现图形界面

其中 zconf.tab.c 程序实现了解析 Kconfig 文件和内核配置主要函数。zconf.tab.c 程序还直接包含了下列一些 C 程序，这样各种配置功能都包含在 zconf.tab.o 目标文件中了。

```
#include "lex.zconf.c"     //lex 语法解析器
#include "util.c"          //配置工具
#include "confdata.c"      //.config 等相关数据文件保存
#include "expr.c"          //表达式函数
#include "symbol.c"        //变量符号处理函数
#include "menu.c"          //菜单控制函数
```

理解这些工具的使用，可以更加方便地配置内核。至于这些工具的源代码实现，一般没有必要去详细分析。

3. Kconfig

Kconfig 文件是 Linux 2.6 内核引入的配置文件，是内核配置选项的源文件。内核源码中的 Documentation/kbuild/kconfig-language.txt 文档有详细说明。

前面已经提到了 arch/$(ARCH)/Kconfig 文件，这是主 Kconfig 文件，跟体系结构有关系。主 Kconfig 文件调用其他目录的 Kconfig 文件，其他的 Kconfig 文件又调用各级子目录的配置文件，成树状关系。

菜单按照树状结构组织，主菜单下有子菜单，子菜单还有子菜单或者配置选项。每个选项可以有依赖关系，这些依赖关系用于确定它是否显示。只有被依赖项父项已经选中，子项才会显示。

下面解释一下 Kconfig 的特点和语法。

（1）菜单项

多数选项定义一个配置选项，其他选项起辅助组织作用。举例说明单个的配置选项的定义。

```
config MODVERSIONS
      bool "Set version information on all module symbols"
      depends MODULES
      help
         Usually, modules have to be recompiled whenever you switch to a new
         kernel.  ...
```

每一行开头用关键字“config”，后面可以跟多行。后面的几行定义这个配置选项的属性。属性包括配置选项的类型、选择提示、依赖关系、帮助文档和缺省值。同名的选项可以重复定义多次，但是每次定义只有一个选择提示并且类型不冲突。

（2）菜单属性

一个菜单选项可以有多种属性，不过这些属性也不是任意用的，受到语法的限制。

每个配置选项必须有类型定义。类型定义包括：bool、tristate、string、hex、int 共 5 种。其中有 2 种基本的类型：tristate 和 string，每种类型定义可以有一个选择提示。表 7.4 说明了菜单的各种属性。

表 7.4 内核菜单属性说明

属　性	语　法	说　明
选择提示	"prompt" <prompt> ["if" <expr>]	每个菜单选项最多有一条提示，可以显示在菜单上。某选择提示可选的依赖关系可以通过“if”语句添加
缺省值	"default" <expr> ["if" <expr>]	配置选项可以有几个缺省值。如果有多个缺省值可选，只使用第一个缺省值。某选项缺省值还可以在其他地方定义，并且被前面定义的缺省值覆盖。如果用户没有设置其他值，缺省值就是配置符号的唯一值。如果有选择提示出现，就可以显示缺省值并且可以配置修改。某缺省值可选的依赖关系可以通过“if”语句添加
依赖关系	"depends on"/"requires" <expr>	这个定义了菜单选项的依赖关系。如果定义多个依赖关系，那么要用“&&”符号连接。依赖关系对于本菜单项中其他所有选项有效（也可以用“if”语句）
反向依赖	"select" <symbol> ["if" <expr>]	普通的依赖关系是缩小符号的上限，反向依赖关系则是符号的下限。当前菜单符号的值用作符号可以设置的最小值。如果符号值被选择了多次，这个限制将被设成最大选择值。反向依赖只能用于布尔或者三态符号
数字范围	"range" <symbol> <symbol> ["if" <expr>]	这允许对 int 和 hex 类型符号的输入值限制在一定范围内。用户输入的值必须大于等于第一个符号值或者小于等于第二个符号值
帮助文档	"help" 或者 "---help---"	这可以定义帮助文档。帮助文档的结束是通过缩进层次判断的。当遇到一行缩进比帮助文档第一行小的时候，就认为帮助文档已经结束。“---help---”和“help”功能没有区别，主要给开发者提供的不同于“help”的帮助

（3）菜单依赖关系

依赖关系定义了菜单选项的显示，也能减少三态符号的选择范围。表达式的三态逻辑比布尔逻辑多一个状态，用来表示模块状态。表 7.5 是菜单依赖关系的语法说明。

表 7.5 菜单依赖关系语法说明

表　达　式	结 果 说 明
<expr> ::= <symbol>	把符号转换成表达式，布尔和三态符号可以转换成对应的表达式值。其他类型符号的结果都是“n”
<symbol> '=' <symbol>	如果两个符号的值相等，返回“y”，否则返回“n”

续表

表　达　式	结 果 说 明
<symbol> '!=' <symbol>	如果两个符号的值相等，返回“n”，否则返回“y”
'(' <expr> ')'	返回表达式的值，括号内表达式优先计算
'!' <expr>	返回(2-/expr/)的计算结果
<expr> '&&' <expr>	返回 min(/expr/, /expr/)的计算结果
<expr> '||' <expr>	返回 max(/expr/, /expr/)的计算结果

一个表达式的值是“n”、“m”或者“y”（或者对应数值的 0、1、2）。当表达式的值为“m”或者“y”时，菜单选项变为显示状态。

符号类型分为两种：常量和非常量符号。

非常量符号最常见，可以通过 config 语句来定义。非常量符号完全由数字符号或者下划线组成。

常量符号只是表达式的一部分。常量符号总是包含在引号范围内的。在引号中，可以使用其他字符，引号要通过“\”号转义。

（4）菜单组织结构

菜单选项的树状结构有两种组织方式。

第一种是显式的声明为菜单。

```
menu "Network device support"
     depends NET
config NETDEVICES
     ...
endmenu
```

“menu”与“endmenu”之间的部分成为“Network device support”的子菜单。所有子选项继承这菜单的依赖关系，例如，依赖关系“NET”就被添加到“NETDEVICES”配置选项的依赖关系列表中。

第二种是通过依赖关系确定菜单的结构。

如果一个菜单选项依赖于前一个选项，它就是一个子菜单。这要求前一个选项和子选项同步地显示或者不显示。

```
config MODULES
     bool "Enable loadable module support"
config MODVERSIONS
     bool "Set version information on all module symbols"
     depends MODULES
comment "module support disabled"
     depends !MODULES
```

MODVERSIONS 依赖于 MODULES，这样只有 MODULES 不是“n”的时候，才显示。反之，MODULES 是“n”的时候，总是显示注释“module support disabled”。

（5）Kconfig 语法

Kconfig 配置文件描述了一系列的菜单选项。每一行都用一个关键字开头（help 文字例外）。菜单的关键字见表 7.6 所示。其中菜单开头的关键字有：config、menuconfig、choice/endchoice、comment、menu/endmenu。它们也可以结束一个菜单选项，另外还有 if/endif、source 也可以结束菜单选项。

表 7.6 Kconfig 菜单关键字说明

关　键　字	语　　法	说　　明
config	"config" <symbol> <config options>	这可以定义一个配置符号<symbol>，并且可以配置选项属性
menuconfig	"menuconfig" <symbol> <config options>	这类似于简单的配置选项，但是它暗示：所有的子选项应该作为独立的选项列表显示
choices	"choice" <choice options> <choice block> "endchoice"	这定义了一个选择组，并且可以配置选项属性。每个选择项只能是布尔或者三态类型。布尔类型只允许选择单个配置选项，三态类型可以允许把任意多个选项配置成“m”。如果一个硬件设备有多个驱动程序，内核一次只能静态链接或者加载一个驱动，但是所有的驱动程序都可以编译为模块。 选择项还可以接受另外一个选项“optional”,可以把选择项设置成“n”，并且不需要选择什么选项
comment	"comment" <prompt> <comment options>	这定义了一个注释，在配制过程中显示在菜单上，也可以回显到输出文件中。唯一可能的选项是依赖关系
Menu	"menu" <prompt> <menu options> <menu block> "endmenu"	这定义了一个菜单项，在菜单组织结构中有些描述。唯一可能的选项是依赖关系
If	"if" <expr> <if block> "endif"	这定义了一个 if 语句块。依赖关系表达式<expr>附加给所有封装好的菜单选项
Source	"source" <prompt>	读取指定的配置文件。读取的文件也会解析生成菜单

7.2.3　Kbuild Makefile

Linux 内核源代码是通过 Makefile 组织编译的。

1．Makefile 的组织结构

Makefiles 包含 5 个部分，见表 7.7 所示。

表 7.7 Makefiles 的 5 个部分

Makefile	顶层目录下的 Makefile
.config	内核配置文件
arch/$(ARCH)/Makefile	对应体系结构的 Makefile

续表

Makefile	顶层目录下的 Makefile
scripts/Makefile.*	所有 kbuild Makefiles 的通用规则等定义
kbuild Makefiles	内核编译各级目录下的 Makefile，大约有 500 多个

顶层目录的 Makefile 读取.config 文件，根据.config 文件中的配置选项编译内核。这个.config 文件是内核配置过程生成的。

顶层目录的 makefile 负责编译 vmlinux（常驻内存的内核映像）和 module（任何模块文件）。它递归地遍历内核源码树中所有子目录，编译所有的目标文件。

编译访问的子目录列表依赖于内核配置。顶层目录的 Makefile 原原本本的包含了一个 arch Makefile（后面将使用这个英文名称），就是 arch/$(ARCH)/Makefile。这个 arch Makefile 给顶层目录提供了体系结构相关的信息。

每个子目录都有一个 Kbuild Makefile（内核编译过程调用），这些 Makefile 执行从上层传递下来的命令。这些 Makefile 使用.config 文件中的信息，构建各种文件列表，由 Kbuild 编译静态链接的或者模块化的目标程序。

scripts/Makefile.*几个文件包含了 Kbuild Makfile 所有的定义和规则等，用于编译内核。

内核源码的大多数 Makefile 是 Kbuild Makefile，使用 Kbuild 组织结构。下面介绍一下 Kbuild Makefile 的语法。

Kbuild 大体上按照下列步骤执行编译过程。

（1）内核配置，生成.config 文件。

（2）保存内核版本信息到 include/linux/version.h。

（3）创建链接符号 include/asm，链接 include/asm-$(ARCH)源目录。

（4）升级所有依赖的前提文件，在 arch/$(ARCH)/Makefile 中指定附加依赖条件。

（5）递归地遍历各级子目录并且编译所有的目标。

init-*、core*、drivers-*、net-*、libs-*的目录变量值在 arch/$(ARCH)/Makefile 文件中有些扩展。

（6）链接所有的目标文件，生成顶层目录的 vmlinux。链接的第一个目标文件在 head-y 列表中，是在 arch/$(ARCH)/Makefile 中定义的。

（7）最后，体系结构相关的部分作必须的后期处理，编译生成最终的引导映像。这可以包括编译引导记录；准备 initrd 映像等类似工作。

2. Makefile 语言

内核 Makefile 是配合 GNU make 使用的。除了 GNU make 的文档中的特点，内核的 Makefile 还有一些 GNU 扩展的功能。

GNU make 支持基本的链接表处理功能。内核 Makefile 使用新颖的编译列表格式，编译过程几乎可以不用 if 语句。

GNU make 有多种变量赋值操作符：“=”、“:=”、“?=”、“+=”。

第 1 种是“=”操作符，在“=”左侧是变量，右侧是变量的值，右侧变量的值可以定义在文件的任何一处，也就是说，右侧中的变量不一定非要是已定义好的值，其也可以使用后

面定义的值。

可以把变量的真实值推到后面来定义。但是这种形式也有不好的地方，那就是递归定义，这会让 make 陷入无限的变量展开过程中去，当然，make 是有能力检测这样的定义，并会报错的。还有就是如果在变量中使用函数，那么，这种方式会让 make 运行时非常慢，更糟糕的是，会使得 wildcard 和 shell 两个函数发生不可预知的错误。因为不会知道这两个函数会被调用多少次。

第 2 种是“:=”操作符，这种方法，前面的变量不能使用后面的变量，只能使用前面已定义好了的变量。如果是这样：

```
y := $(x) bar
x := foo
```

那么，y 的值是“bar”，而不是“foo bar”。

第 3 种是“?=”操作符，先看示例：

```
FOO ?= bar
```

其含义是：如果 FOO 没有被定义过，那么变量 FOO 的值就是“bar”；如果 FOO 先前被定义过，那么这条语将什么也不做。

第 4 种是“+=”操作符，将右边的变量值附加给左边的变量。例如：

```
FOO = string1
FOO += string2
```

这时，FOO 的变量值为“string1 string2”。

3．Kbuild 变量

顶层 Makefile 输出下列变量。

（1）VERSION, PATCHLEVEL, SUBLEVEL, EXTRAVERSION 定义了当前内核版本。

$(VERSION)、$(PATCHLEVEL)和$(SUBLEVEL)定义了基本的 3 个版本号，例如：2、6 和 14，都是数字，对应内核版本号。

$(EXTRAVERSION)为预先或者附加的补丁定义了更细的子版本号。通常是非数字的字符串或者空的，例如：-mm1。

（2）KERNELRELEASE 定义了内核发布的版本，一般是单个的字符串，例如：2.6.14。常用来作为版本显示。

（3）ARCH 定义了目标板体系结构，例如：i386、arm 或者 sparc。一些 kbuild Makefile 测试$(ARCH)来确定要编译哪一个文件。顶层 Makefile 缺省地把$(ARCH)设置成主机系统的体系结构。对于交叉编译，需要修改定义或者在命令行重载这个值。例如：

```
make ARCH=arm
```

（4）INSTALL_PATH 为 arch Makefile 定义了安装驻留内存的内核映像和 System.map 文件。使用这个体系结构安装目标板。

（5）INSTALL_MOD_PATH 和 MODLIB。

$(INSTALL_MOD_PATH)在安装模块的时候作为$(MODLIB)的前缀。这个变量没有在 Makefile 中定义，但是可以通过命令行传递。

$(MODLIB)指定模块安装的路径。顶层 Makefile 的$(MODLIB)缺省定义如下。

$(INSTALL_MOD_PATH)/lib/modules/$(KERNELRELEASE)

4．Kbuild Makefile 的定义

（1）目标定义

目标定义 kbuild Makefile 的核心。它们定义了要编译的文件、特殊的编译选项和要递归地遍历的子目录。

最简单的 kbuild makefile 包含一行，例如：

```
obj-y += foo.o
```

这是告诉 kbuild，当前目录中要编译一个目标文件 foo.o，foo.o 应该从 foo.c 或者 foo.S 编译过来。

如果要把 foo.o 编译为模块，就使用变量 obj-m。因此，经常用下列方式：

```
obj-$(CONFIG_FOO) += foo.o
```

$(CONFIG_FOO)可以配置为 y（静态链接）或者 m（动态模块）。如果 CONFIG_FOO 即不是 y，也不是 m，那么这个文件就不被编译或者链接。

（2）静态链接目标文件- obj-y

kbuild Makefile 指定了 vmlinux 的目标文件，就在$(obj-y)列表中。这些列表依赖于内核的配置。

Kbuild 编译所有的$(obj-y)文件，再用$(LD)命令把目标文件链接成一个 built-in.o 文件。然后 built-in.o 将被链接到顶层目录的 vmlinux 中去。

$(obj-y)中的文件顺序很重要。因为列表中允许重复，第一个实例链接到 built-in 中以后，后面的实例将被忽略。另外，某些函数（module_init() / __initcall）会在启动过程中按照排列顺序调用。如果改变链接顺序，也可能改变设备的初始化顺序。例如：改变 SCSI 控制器的探测顺序，就会导致磁盘重复编号。

举例说明 obj-y。

```
#drivers/isdn/i4l/Makefile
# Makefile for the kernel ISDN subsystem and device drivers.
# Each configuration option enables a list of files.
obj-$(CONFIG_ISDN)                        += isdn.o
obj-$(CONFIG_ISDN_PPP_BSDCOMP)               += isdn_bsdcomp.o
```

（3）可加载模块目标文件- obj-m

$(obj-m)用来指定要编译成可加载模块的目标文件。

一个模块可以由一个或者几个源文件编译生成。对于单个源文件的情况，kbuild Makefile 可以简单地把文件添加到$(obj-m)中即可。

例如：

```
#drivers/isdn/i4l/Makefile
obj-$(CONFIG_ISDN_PPP_BSDCOMP) += isdn_bsdcomp.o
```

注意 这里的$(CONFIG_ISDN_PPP_BSDCOMP)配置为“m”。

如果内核模块由几个源文件编译生成，要指定要编译成一个模块。Kbuild 需要知道这个模块包含哪些目标文件，那么必须设置一个$(<module_name>-objs)变量。

例如：

```
#drivers/isdn/i4l/Makefile
obj-$(CONFIG_ISDN) += isdn.o
isdn-objs := isdn_net_lib.o isdn_v110.o isdn_common.o
```

在这个例子中，模块的名字是 isdn.o。Kbuild 会编译$(isdn-objs)列表中的目标文件，然后执行“$(LD) –r”命令，把这些目标文件链接成 isdn.o。

Kbuild 可以通过-objs 和-y 后缀识别组成复合目标的目标文件。这允许 Makefile 通过 CONFIG_符号的值来确定一个目标文件是不是一个复合目标文件。

例如：

```
#fs/ext2/Makefile
obj-$(CONFIG_EXT2_FS)        += ext2.o
ext2-y                       := balloc.o bitmap.o
ext2-$(CONFIG_EXT2_FS_XATTR) += xattr.o
```

这个例子中，如果$(CONFIG_EXT2_FS_XATTR)配置为“y”，xattr.o 就是复合目标 ext2.o 的一部分。

注意 当把这些目标文件编译链接到内核中去的时候，上面的语法仍然有效。如果配置 CONFIG_EXT2_FS=y，kbuild 会单独编译 ext2.o 文件，再链接到 built-in.o 文件中。

（4）库目标文件 lib-y

用 obj-列出的目标文件可以用于模块或者指定目录的 built-in.o 文件，也可以列出要包含到一个库 lib.a 中的目标文件。用 lib-y 列出的目标文件可以组合到目录下的一个库中。在 obj-y 中列出并且在 lib-y 中列出的目标文件不会包含到这个库中，因为它们是内核可以访问的。为了一致性，在 lib-m 中列出的目标文件会包含到 lib.a 中。

注意 同一个 kbuild makefile 可以把文件列到 built-in 表中，同时还是一个库的列表的一部分。因此，同一个目录可以包含一个 built-in.o 和一个 lib.a 文件。

例如：

```
#arch/i386/lib/Makefile
lib-y    := checksum.o delay.o
```

这会基于 checksum.o 和 delay.o 创建一个 lib.a 文件。为了让 kbuild 知道有一个 lib.a 要编译，这个目录应该添加到 libs-y 列表中。

lib-y 一般仅限于 lib/和 arch/*/lib/目录。

（5）遍历子目录

一个 Makefile 只负责在自己的目录下编译目标文件。各子目录下的文件应该由各自的 Makefile 来管理。编译系统会自动在子目录中递归地调用 make。

这项工作也要用到 obj-y 和 obj-m。比如 ext2 在一个单独的目录中，在 fs 目录下的 Makefile 使用下面的配置方法。

```
#fs/Makefile
obj-$(CONFIG_EXT2_FS) += ext2/
```

如果 CONFIG_EXT2_FS 设成“y”或者“m”，对应的 obj-变量就会设置，并且 Kbuild 就会下到 ext2 目录中编译。Kbuild 只使用这个信息决定是否需要访问这个目录，编译的工作是子目录中的 Makefile 负责。

对于 CONFIG_选项既不是“y”也不是“m”的目录，使用 CONFIG_变量可以让 Kbuild 忽略掉。这是一个很好的办法。

（6）编译标志

编译标志包括：EXTRA_CFLAGS、EXTRA_AFLAGS、EXTRA_LDFLAGS、EXTRA_ARFLAGS。

所有的 EXTRA_变量只适用于当前的 Kbuild makefile。EXTRA_变量适用 Kbuild makefile 中执行的所有的命令。

$(EXTRA_CFLAGS)通过$(CC)指定编译 C 文件的选项。

例如：

```
# drivers/sound/emu10k1/Makefile
EXTRA_CFLAGS += -I$(obj)
ifdef DEBUG
     EXTRA_CFLAGS += -DEMU10K1_DEBUG
endif
```

因为顶层目录 Makefile 的变量$(CFLAGS)用于整个源码树的编译，所以这种变量定义是必须的。

在编译汇编语言源码的时候，$(EXTRA_AFLAGS)是与每个目录的选项类似的字符串。

例如：

```
#arch/x86_64/kernel/Makefile
EXTRA_AFLAGS := -traditional
```

$(EXTRA_LDFLAGS)和$(EXTRA_ARFLAGS)分别是与每个目录$(LD)和$(AR)的选项类似的字符串。

例如：

```
#arch/m68k/fpsp040/Makefile
EXTRA_LDFLAGS := -x
```

CFLAGS_$@, AFLAGS_$@

CFLAGS_$@和 AFLAGS_$@仅适用于当前 kbuild makefile 的命令。

$(CFLAGS_$@)为$(CC)指定每个文件的选项。$@代表指定的文件名。

例如：

```
# drivers/scsi/Makefile
CFLAGS_aha152x.o =   -DAHA152X_STAT -DAUTOCONF
CFLAGS_gdth.o     = # -DDEBUG_GDTH=2 -D__SERIAL__ -D__COM2__ \
             -DGDTH_STATISTICS
CFLAGS_seagate.o =  -DARBITRATE -DPARITY -DSEAGATE_USE_ASM
```

这些行指定了 aha152x.o、gdth.o 和 seagate.o 的编译标志。

$(AFLAGS_$@)对于汇编语言编译有类似的特点。

例如：

```
# arch/arm/kernel/Makefile
AFLAGS_head-armv.o := -DTEXTADDR=$(TEXTADDR) -traditional
AFLAGS_head-armo.o := -DTEXTADDR=$(TEXTADDR) -traditional
```

（7）依赖跟踪

Kbuild 按照下列步骤跟踪依赖关系。

- 所有依赖的前提文件（*.c 和*.h 文件）
- 在依赖的前提文件中用到的 CONFIG_选项
- 编译目标文件用到的命令行

因此，如果修改$(CC)的一个选项，所有相关的文件都会重新编译。

（8）特殊的规则

当 Kbuild 结构不提供必须的支持的时候，要使用特殊规则。一个典型的例子是在编译过程中生成头文件。另外一个例子是体系结构相关的 Makefile 需要特殊的规则准备映像等。

特殊的规则可以按照普通的规则来写。Kbuild 不在 Makefile 所在的目录执行，因此所有特殊的规则应该为依赖的文件和目标文件提供相对路径。

定义特殊规则的时候，常用到两个变量：$(src)和$(obj)。

$(src)是指向 Makefile 所在的目录的相对路径。当引用位于源码树的文件的时候，总是使用$(src)。

$(obj)是指向目标文件保存的相对路径。当引用生成文件的时候，总是使用$(obj)。

例如：

```
#drivers/scsi/Makefile
$(obj)/53c8xx_d.h: $(src)/53c7,8xx.scr $(src)/script_asm.pl
      $(CPP) -DCHIP=810 - < $< | ... $(src)/script_asm.pl
```

这是一个特殊规则，遵守普通的 make 语法 uo$(src)前缀。

5. 体系结构相关的 Makefile 定义

顶层的 Makefile 在开始遍历各级子目录之前，要设置环境变量和做准备工作。

顶层目录 Makefile 包含通用的部分，arch/$(ARCH)/Makefile 则包含了设置 Kbuild 指定的体系结构需要的东西。因此，arch/$(ARCH)/Makefile 设置一些变量并且定义一些目标规则。

（1）通过变量设置编译体系结构相关代码

LDFLAGS_vmlinux 是用来指定 vmlinux 额外的编译标志，在链接最终的 vmlinux 时传递给链接器，通过 LDFLAGS_$@调用。

例如：

```
#arch/arm/Makefile
LDFLAGS_vmlinux     :=-p --no-undefined -X
```

CFLAGS 是$(CC)编译选项标志。缺省值在顶层 Makefile 中定义。对于不同的体系结构，有附加选项。通常 CFLAGS 变量依赖于内核配置。

例如：

```
arch-$(CONFIG_CPU_32v4)     :=-D__LINUX_ARM_ARCH__=4 -march=armv4
CFLAGS        +=$(CFLAGS_ABI) $(arch-y) $(tune-y)
```

许多体系结构相关的 Makefile 通过目标板 C 编译器动态地探测支持选项。比如可以把相关选项中的配置选项扩展为“y”。

CFLAGS_KERNEL 是$(CC)编译 built-in 的专用选项。它包含了用于编译驻留内存内核代码的额外 C 编译标志。

CFLAGS_MODULE 是$(CC)编译模块专用选项。它包含了用于编译可动态加载的内核模块的 C 编译选项。

（2）添加 archprepare 规则的依赖条件

archprepare 规则用来列出编译依赖的前提条件，在开始进入各级子目录编译之前，先生成依赖的文件。例如：

```
#arch/arm/Makefile
archprepare: maketools
```

这个例子中，在进入子目录编译之前，要先处理 maketools 文件。许多头文件的生成也使用 archprepare 规则。

（3）列出要遍历的子目录

arch Makefile 配合顶层目录的 Makefile 定义如何编译 vmlinux 的变量。对于模块没有对应体系结构的定义，所以模块编译方法是和体系结构无关的。

编译列表包括：head-y，init-y，core-y，libs-y，drivers-y，net-y。

$(head-y)列出链接到 vmlinux 的起始位置的目标文件。

$(libs-y)列出 lib.a 的库文件所在的目录。

剩余列出的目录都是 built-in.o 文件所在的目录。

链接过程中，$(init-y)列出的目标文件紧跟在$(head-y)后面。然后是$(core-y)、$(libs-y)、$(drivers-y)和$(net-y)。

顶层目录 Makefile 的定义包含了所有普通目录，arch/$(ARCH)/Makefile 只添加体系结构相关的目录。

例如：

```
#arch/arm/Makefile
core-y              += arch/arm/kernel/ arch/arm/mm/ arch/arm/common/
core-y              += $(MACHINE)
libs-y          += arch/arm/lib/
drivers-$(CONFIG_OPROFILE)     += arch/arm/oprofile/
```

（4）体系结构相关的映像

arch Makefile 还定义 vmlinux 文件编译的规则，并且压缩打包到自引导代码中，在相应的目录下生成 zImage。这里包括各种不同的安装命令。不同的体系结构没有标准的规则。

通常是在 arch/$(ARCH)/boot 目录下做一些特殊处理。

Kbuild 不负责支持 arch/$(ARCH)/boot 目录的编译。因此，arch/$(ARCH)/Makefile 文件应该自己定义编译目标。

推荐的方法是包含 arch/$(ARCH)/Makefile 中包含快捷方式，使用全路径调用子目录下的 Makefile。

例如：

```
#arch/arm/Makefile
boot := arch/arm/boot
zImage Image xipImage bootpImage uImage: vmlinux
     $(Q)$(MAKE) $(build)=$(boot) MACHINE=$(MACHINE) $(boot)/$@
```

“(Q)(MAKE) $(build)=<dir>”是在一个子目录<dir>中调用 make 的推荐方法。

在这里，不同体系结构的相关目标定义没有一致的规则，可以通过“make help”命令列出相关的帮助。因此，还要定义帮助信息。

例如：

```
#arch/arm/Makefile
define archhelp
  echo  '* zImage       - Compressed kernel image (arch/$(ARCH)/boot/zImage)'
……
endef
```

当不带参数执行 make 的时候，首先会编译遇到的第一个目标。顶层目录 Makefile 中的第一个目标是 all。不同体系结构应该定义缺省的引导映像，在“make help”中加注*号，而且加到目标 all 的前提条件中。

例如：

```
#arch/arm/Makefile
# Default target when executing plain make
ifeq ($(CONFIG_XIP_KERNEL),y)
all: xipImage
else
all: zImage
endif
```

当配置好了内核以后，执行 make。如果没有把内核配置成 XIP 方式，就调用 zImage 的规则。

（5）编译非 kbuild 目标

除了使用 obj-*列表指定编译的目标文件以外，还可以使用 extra-y 列表指定当前目录下要创建的附加目标。

对于以下两种情况需要 extra-y 列表。

- 使 kbuild 在命令行中检查文件修改变化。比如使用$(call if_changed,xxx)语句。
- 告诉 kbuild 要要编译或者删除哪些文件。

例如：

```
#arch/arm/kernel/Makefile
extra-y := head.o init_task.o
```

在这个例子中，extra-y 用来列出应该编译的目标文件，但是不应该连接到 built-in.o 中。

（6）编译自引导映像有用的命令

kbuild 提供了一些编译引导映像时很有用的宏。

- if_changed

if_changed 是下列命令的基本构成部分。

```
target: source(s) FORCE
      $(call if_changed,ld/objcopy/gzip)
```

编译这个规则的时候，首先检查是否有文件需要更新或者命令行有没有改变。如果任何编译选项改变，会强制重新编译。任何使用 if_changed 的目标必须列在$(targets)中，否则命令行检查会出错，并且目标总是会编译。

注意 一个常见的错误是忘记 FORCE 前提条件。

另外一个问题是有无空格很重要。例如：下列语句的逗号后面有一个空格，这个空格会导致语法错误。

```
target: source(s) FORCE
      $(call if_changed, ld/objcopy/gzip)   #WRONG!#
```

- ld

具有链接目标的功能。通常使用 LDFLAGS_$@设置 ld 的选项。

- objcopy

复制转换二进制程序。使用在 arch/$(ARCH)/Makefile 中的 OBJCOPYFLAGS 编译选项。OBJCOPYFLAGS_$@可以用来添加附加的编译选项。

- gzip

压缩目标。使用最大压缩方式。

例如：

```
$(obj)/piggy.gz: $(obj)/../Image FORCE
      $(call if_changed,gzip)
```

（7）定制 kbuild 命令

当 kbuild 带 KBUILD_VERBOSE=0 选项执行的时候，通常只会显示一个命令的简写。要在自定义的 kbuild 命令中使能这种功能，必须设置以下两个变量。

- quiet_cmd_<command> 代表要显示的命令
- cmd_<command> 代表要执行的命令

例如：

```
#arch/arm/boot/Makefile
quiet_cmd_uimage = UIMAGE  $@
       cmd_uimage = $(CONFIG_SHELL) $(MKIMAGE) -A arm -O linux -T kernel \
             -C none -a $(ZRELADDR) -e $(ZRELADDR) \
             -n 'Linux-$(KERNELRELEASE)' -d $< $@
$(obj)/uImage:  $(obj)/zImage FORCE
      $(call if_changed,uimage)
      @echo '  Image $@ is ready'
```

当带“KBUILD_VERBOSE=0”更新编译的时候，只显示下列一行，而不会把编译信息都显示出来。

```
UIMAGE  arch/arm/boot/uImage
```

（8）预处理链接脚本

当编译 vmlinux 映像的时候，将用到链接脚本 arch/$(ARCH)/kernel/vmlinux.lds。这个脚本的预处理变体文件是相同目录下的 vmlinux.lds.S。

Kbuild 知道.lds 文件并且包含*.lds.S 到*.lds 的转换规则。

例如：

```
#arch/i386/kernel/Makefile
always := vmlinux.lds
```

$(always)列表告诉 kbuild 编译目标 vmlinux.lds。

```
#Makefile
export CPPFLAGS_vmlinux.lds += -P -C -U$(ARCH)
```

$(CPPFLAGS_vmlinux.lds)列表告诉 kbuild 在编译 vmlinux.lds 的时候使用指定的选项。

当编译*.lds 目标文件的时候，kbuild 使用以下变量。

- CPPFLAGS：在顶层目录 Makefile 中定义。
- EXTRA_CPPFLAGS：可以在 kbuild Makefile 中定义。
- CPPFLAGS_$(@F)：目标板特定选项。

Kbuild 的*.lds 文件结构在多种体系结构的文件中使用。

（9）$(CC)支持的函数

内核编译可能会使用不同版本的$(CC)，不同版本支持独立的一套选项和特点。Kbuild 提供检查$(CC)有效选项的基本支持。$(CC)一般就是 gcc 编译器，但是可能会有其他替代。

- cc-option

cc-option 选项用于检查$(CC)是否支持一个给定的选项或者第二个可选项。

例如：

```
#arch/i386/Makefile
cflags-y += $(call cc-option,-march=pentium-mmx,-march=i586)
```

在上面的例子中，如果$(CC)支持-march=pentium-mmx，那么 cflags-y 会被赋给这个选项。否则，使用-march-i586 选项。如果没有后一个选项，在前一个选项不支持的情况下，cflags-y 不会赋什么值。

- cc-option-yn

cc-option-yn 用于检查 gcc 是否支持给定的选项。如果支持，返回“y”，否则返回“n”。

例如：

```
#arch/ppc/Makefile
biarch := $(call cc-option-yn, -m32)
aflags-$(biarch) += -a32
cflags-$(biarch) += -m32
```

在上面的例子中，如果$(CC)支持-m32 选项，$(biarch)就设成“y”。当$(biarch)等于“y”时，扩展变量$(aflags-y)和$(cflags-y)会赋值-a32 和-m32。

- cc-option-align

gcc 版本大于等于 3.0 时，使用选项的移位类型指定函数的对齐。用作选项前缀的$(cc-option-align)会选择合适的前缀。

cc-option-align 的伪语言描述如下。

```
if gcc < 3.00
       cc-option-align = -malign
else if gcc >= 3.00
       cc-option-align = -falign
```

例如：

```
CFLAGS += $(cc-option-align)-functions=4
```

上面的例子，对于 gcc>=3.00，选项为-falign-functions=4；对于 gcc<3.00，选项为-malign-functions=4。

- cc-version

cc-version 返回$(CC)编译器的版本号数字。格式为代表主从版本号的 2 个十进制数。例如：gcc 3.41 应该返回 0341。

当$(CC)版本在特定区域会导致错误的时候，cc-version 是很有用的。例如：-mregparm=3 选项的支持在一些 gcc 版本中不完整。

例如：

```
#arch/i386/Makefile
GCC_VERSION := $(call cc-version)
cflags-y += $(shell \
if [ $(GCC_VERSION) -ge 0300 ] ; then echo "-mregparm=3"; fi ;)
```

上面的例子中，-mregparm=3 只能用于版本大于等于 3.0 的 gcc。

7.2.4 内核编译

1. 编译命令

Makefile 还提供了配置编译的选项或者规则。执行 make help，可以打印出详细的帮助信息。解释一下帮助信息列出的各种选项的含义，分别在每一行信息下面加以注释。

```
$make help
```

打印出下列帮助信息。

（1）用于清理生成文件的目标（Cleaning targets）

```
clean          - remove most generated files but keep the config
```

clean 目标可以清除大多数生成的文件，但是保留.config。

```
mrproper       - remove all generated files + config + various backup files
```

mrproper 可以清除所有生成的文件，包括.config 和各种备份文件。

（2）内核配置的目标（Configuration targets）

```
config         - Update current config utilising a line-oriented program
```

config 是命令行的内核配置方式。

```
menuconfig     - Update current config utilising a menu based program
```

menuconfig 是光标菜单内核配置方式。

```
xconfig         - Update current config utilising a QT based front-end
```

xconfig 是基于 QT 图形界面的内核配置方式。

```
gconfig         - Update current config utilising a GTK based front-end
```

gconfig 是基于 GTK 图形界面的内核配置方式

```
oldconfig       - Update current config utilising a provided .config as base
```

oldconfig 基于已有的.config 文件进行内核配置。

```
randconfig      - New config with random answer to all options
```

randconfig 是对所有的选项按照随机回答（Y/M/N）的方式生成新配置。

```
defconfig       - New config with default answer to all options
```

defconfig 是对所有的选项都按照缺省回答生成新配置。

```
allmodconfig    - New config selecting modules when possible
```

allmodconfig 是对所有选项尽可能配置模块的新配置。

```
allyesconfig    - New config where all options are accepted with yes
```

allyesconfig 是对所有选项都配置成“Yes”的最大配置。

```
allnoconfig     - New minimal config
```

allnoconfig 是对所有选项都配置成“No”的最小配置。
（3）其他通用目标（Other generic targets）

```
all             - Build all targets marked with [*]
```

all 是编译所有标记星号的目标，也就是编译所有缺省目标。

```
* vmlinux       - Build the bare kernel
```

vmlinux 是编译最基本的内核映像，就是顶层的 vmlinux。

```
* modules       - Build all modules
```

modules 是编译所有的模块。

```
modules_install - Install all modules
```

modules_install 是安装所有的模块。

```
dir/            - Build all files in dir and below
```

dir 是编译 dir 目录及其子目录的所有文件，当然 dir 代表具体的一个目录名。

```
dir/file.[ois]  - Build specified target only
```

dir/file.[ois]是仅编译 dir 目录下指定的目标。

```
dir/file.ko     - Build module including final link
```

dir/file.ko 是编译并且链接指定目录的模块。

```
rpm             - Build a kernel as an RPM package
```

rpm 是以 RPM 包方式编译内核。

```
tags/TAGS       - Generate tags file for editors
```

tags/TAGS 是为编辑器生成 tag 文件，方便编辑器识别关键词。

```
cscope          - Generate cscope index
```

cscope 是生成 cscope 索引，方便代码浏览。

```
kernelrelease   - Output the release version string
```

kernelrelease 是输出内核版本的字符串。

（4）静态解析器（Static analysers）

```
buildcheck      - List dangling references to vmlinux discarded sections
                  and init sections from non-init sections
```

buildcheck 是列出对 vmlinux 废弃段的虚引用和从非 init 段引用 init 段的虚引用。

```
checkstack      - Generate a list of stack hogs
```

checkstack 是生成栈空间耗费者的列表。

```
namespacecheck  - Name space analysis on compiled kernel
```

namespace 是对编译好的内核做命名域分析。

（5）内核打包（Kernel packaging）

```
rpm-pkg         - Build the kernel as an RPM package
```

rpm-pkg 是以一个 RPM 包的方式编译内核。

```
binrpm-pkg      - Build an rpm package containing the compiled kernel
                  and modules
```

binrpm-pkg 是编译一个包含已经编译好的内核和模块的 rpm 包。

```
deb-pkg         - Build the kernel as an deb package
```

deb-pkg 是以一个 deb 包的方式编译内核。

```
tar-pkg         - Build the kernel as an uncompressed tarball
```

tar-pkg 是以一个不压缩的 tar 包方式编译内核。

```
targz-pkg       - Build the kernel as a gzip compressed tarball
```

targz-pkg 是以一个 gzip 压缩包的方式编译内核。

```
tarbz2-pkg      - Build the kernel as a bzip2 compressed tarball
```

tarbz2-pkg 是以一个 bzip2 压缩包的方式编译内核。

（6）文档目标（Documentation targets）

```
Linux kernel internal documentation in different formats:
xmldocs (XML DocBook), psdocs (Postscript), pdfdocs (PDF)
htmldocs (HTML), mandocs (man pages, use installmandocs to install)
```

Linux 内核内部支持各种形式的文档。

（7）体系结构相关的目标（ARM）（Architecture specific targets (arm)）

```
* zImage        - Compressed kernel image (arch/arm/boot/zImage)
```

zImage 是编译生成压缩的内核映像（arch/arm/boot/zImage）。

```
Image         - Uncompressed kernel image (arch/arm/boot/Image)
```

Image 是编译生成非压缩的内核映像（arch/arm/boot/Image）。

```
* xipImage      - XIP kernel image, if configured (arch/arm/boot/xipImage)
```

xipImage 是编译生成 XIP 的内核映像（arch/arm/boot/xipImage），前提是内核配置成 XIP。

```
  bootpImage    - Combined zImage and initial RAM disk
                  (supply initrd image via make variable INITRD=<path>)
```

bootpImage 是编译包含 zImage 和 initrd 的映像（可以通过 make 变量 INITRD=<path>提供 initrd 映像）。

```
install       - Install uncompressed kernel
```

install 是安装非压缩的内核。

```
zinstall      - Install compressed kernel
```

```
                          Install using (your) ~/bin/installkernel or
                          (distribution) /sbin/installkernel or
                          install to $(INSTALL_PATH) and run lilo
```

zinstall 是安装压缩的内核。通过发行版的/bin/installkernel 工具安装，或者安装到$(INSTALL_PATH)路径下，然后再执行 lilo。这些目标对于 X86 平台适用。

还有各种开发板的缺省内核配置文件，这些配置文件都保存在 arch/arm/configs 目录下。

```
  assabet_defconfig       - Build for assabet
……
  smdk2410_defconfig       - Build for smdk2410
  spitz_defconfig          - Build for spitz
  versatile_defconfig     - Build for versatile
```

每种支持的目标板都会保存一个缺省的内核配置文件。

```
make V=0|1 [targets] 0 => quiet build (default), 1 => verbose build
```

V=0 表示不显示编译信息（缺省），V=1 表示显示编译信息。

```
make O=dir [targets] Locate all output files in "dir", including .config
```

O=dir 用来指定所有输出文件的目录，包括.config 文件，都将放到 dir 目录下。

```
make C=1   [targets] Check all c source with $CHECK (sparse)
```

C=1 表示检查所有$CHECK 的 C 程序。

```
make C=2   [targets] Force check of all c source with $CHECK (sparse)
```

C=2 表示强制检查所有$CHECK 的 C 程序。

```
Execute "make" or "make all" to build all targets marked with [*]
```

执行 make 或者 make all，将自动编译所有带星号标志的目标。

```
For further info see the ./README file
```

更多信息参看 REAME 文件。

其中，vmlinux modules zImage 和 xipImage 是 Makefile 缺省的目标。执行 make，缺省地就可以执行这些编译规则。但是，zImage 和 xipImage 是互斥的，因为两种内核映像格式不可能同时配置。

2．编译链接内核映像

一般情况下，先编译链接生成顶层目录的 vmlinux，再把 vmlinux 精简压缩成 piggy.gz，然后加上自引导程序链接成 arch/$(ARCH)/boot/zImage，这样就得到一个具备自启动能力的

Linux 内核映像。

除了 zImage 之外，还有其他一些映像格式，分别适用于不同的体系结构和引导程序。

由于 zImage 是最通用的，所以我们只分析一下 zImage 编译链接的过程。

（1）编译链接 vmlinux

vmlinux 的规则是在顶层的 Makefile 中定义的。vmlinux 是由$(vmlinux-init)和$(vmlinux-main)列表中指定的目标文件链接而成的，大多数是来自顶层子目录下的 built-in.o 文件，其他都在 arch/$(ARCH)Makefile 中指定。这些目标文件的链接顺序非常重要，$(vmlinux-init)必须排在第一位。参考 Makefile 注释中的结构图。

vmlinux 的版本（uname-v 可以显示）不是在各级目录的编译阶段更新的，因为还不知道是否需要更新 vmlinux。除了在添加内核符号之前生成 kallsysms 信息的情况，直到链接 vmlinux 才更新 vmlinux 版本信息。还生成 System.map 文件，用来描述所有符号（全局变量、函数等）的地址。

```
#Makefile
#
# vmlinux
#   ^
#   |
#   +-< $(vmlinux-init)
#   |   +--< init/version.o + more
#   |
#   +--< $(vmlinux-main)
#   |    +--< driver/built-in.o mm/built-in.o + more
#   |
#   +-< kallsyms.o (see description in CONFIG_KALLSYMS section)
#
vmlinux-init := $(head-y) $(init-y)
vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)
vmlinux-all  := $(vmlinux-init) $(vmlinux-main)
vmlinux-lds  := arch/$(ARCH)/kernel/vmlinux.lds

# Rule to link vmlinux - also used during CONFIG_KALLSYMS
# May be overridden by arch/$(ARCH)/Makefile
quiet_cmd_vmlinux__ ?= LD     $@
      cmd_vmlinux__ ?= $(LD) $(LDFLAGS) $(LDFLAGS_vmlinux) -o $@ \
      -T $(vmlinux-lds) $(vmlinux-init)                    \
      --start-group $(vmlinux-main) --end-group             \
      $(filter-out $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) FORCE ,$^)
```

这里通过定制 Kbuild 命令来定义 vmlinux 的规则。cmd_vmlinux__命令就是具体链接生

成 vmlinux 的 Kbuild 命令。命令各部分的具体含义如下。

$(LD)是链接工具，对于 ARM 平台，就是 arm-linux-ld；

$(LDFLAGS)和$(LDFLAGS_vmlinux)是链接选项列表；

“-o $@”选项指定了生成的文件，$@在编译 vmlinux 目标的时候，代表 vmlinux 文件；

“-T $(vmlinux-lds)”选项指定链接脚本，arch/$(ARCH)/kernel/vmlinux.lds 就是这个脚本，vmlinux-lds 是在 Makefile 中定义的。

“$(vmlinux-init)”是初始化部分目标文件列表，放在开头。

“--start-group $(vmlinux-main) --end-group”是内核主要的目标文件，链接的时候要检查函数等符号是否确定。

剩余的代码或者数据链接在最后。

（2）生成 vmlinux.lds 链接脚本

vmlinux 的链接脚本是 arch/$(ARCH)/kernel/vmlinux.lds。Kbuild 可以根据模板 vmlinux.lds.S 转换生成。在 scripts/Makefile.build 中定义了一条把.lds.S 转换成.lds 的规则。

```
#scripts/Makefile.build
# Linker scripts preprocessor (.lds.S -> .lds)
# ---------------------------------------------------
quiet_cmd_cpp_lds_S = LDS     $@
      cmd_cpp_lds_S = $(CPP) $(cpp_flags) -D__ASSEMBLY__ -o $@ $<
%.lds: %.lds.S FORCE
      $(call if_changed_dep,cpp_lds_S)

摘取 arch/arm/kernel/vmlinux.lds.S 的部分内容，解释一下。

/* arch/arm/kernel/vmlinux.lds.S */
/* 由于使用了一些宏，所以需要包含这几个宏定义的头文件 */
#include <asm-generic/vmlinux.lds.h>
#include <linux/config.h>
#include <asm/thread_info.h>

OUTPUT_ARCH(arm)            /* 指定目标板体系结构 */
ENTRY(stext)                /* 代码段入口 */
SECTIONS                    /* 代码段各部分 */
{
    . = TEXTADDR;           /* 代码段起始地址，大多数 Linux 内核是 0xC0008000 */
    .init : {               /* 内核初始化的代码和数据 */
            _stext = .;
                 _sinittext = .;
                 *(.init.text)
                 _einittext = .;
```

```
            __proc_info_begin = .;
                *(.proc.info.init)
            __proc_info_end = .;
            __arch_info_begin = .;
                *(.arch.info.init)
            __arch_info_end = .;
            ……
    }

    /DISCARD/ : {              /* 内核退出的代码和数据 */
        *(.exit.text)
        *(.exit.data)
        *(.exitcall.exit)
    }

    .text : {              /* 真正的代码段部分 */
           _text = .;      /* 代码和只读数据 */
                *(.text)
                SCHED_TEXT
                LOCK_TEXT
                *(.fixup)
                *(.gnu.warning)
                *(.rodata)
                *(.rodata.*)
                *(.glue_7)
                *(.glue_7t)
        *(.got)               /* Global offset table */
    }
    RODATA
    _etext = .;        /* 代码段和只读数据结束 */
    ……

    .data : AT(__data_loc) { /* 数据段起始 */
          __data_start = .;/* 内存中的地址 */
          ……
          /* 例外修正表（可能需要在运行时修正） */
          . = ALIGN(32);
          __start___ex_table = .;
          *(__ex_table)
```

```
            __stop___ex_table = .;

            /* 普通的数据段 */
            *(.data)
            CONSTRUCTORS
            _edata = .;      /* 数据段结束 */
      }

      .bss : { /* 未初始化的全局变量 */
             __bss_start = .;/* BSS */
             *(.bss)
             *(COMMON)
             _end = .;
      }

      /* 调试信息和数据段.*/
      .stab 0 : { *(.stab) }
      .stabstr 0 : { *(.stabstr) }
      .stab.excl 0 : { *(.stab.excl) }
      .stab.exclstr 0 : { *(.stab.exclstr) }
      .stab.index 0 : { *(.stab.index) }
      .stab.indexstr 0 : { *(.stab.indexstr) }
      .comment 0 : { *(.comment) }
}
```

vmlinux 程序的链接组装，就是完全按照上面脚本的顺序。这样，内核代码和数据才能加载到相应的位置运行。

（3）链接生成 zImage

zImage 的规则是在 arch/$(ARCH)Makefile 中定义的，它总是与目标板体系结构有关。

```
#arch/arm/Makefile
zImage Image xipImage bootpImage uImage: vmlinux
    $(Q)$(MAKE) $(build)=$(boot) MACHINE=$(MACHINE) $(boot)/$@
```

zImage 的前提条件是 vmlinux，也就是说，只有顶层的 vmlinux 编译通过，才能生成 zImage。

编译命令是到 arch/$(ARCH)/boot 目录下，调用 Makefile 的 zImage 规则，同时传递变量 MACHINE。其中$@就是 zImage，这个规则又在 arch/arm/boot/Makefile 中定义。

```
#arch/arm/boot/Makefile
$(obj)/zImage:  $(obj)/compressed/vmlinux FORCE
```

```
        $(call if_changed,objcopy)
        @echo '  Kernel: $@ is ready'
```

这条规则的前提条件是$(obj)/compressed/vmlinux，那么这又要编译子目录 compressed。事实上，对于不同的体系结构，这部分代码有很大差异。对于 ARM 平台来说，$(obj)/compressed/vmlinux 就是压缩的自引导映像，只不过没有精简。

在 arch/arm/boot/compressed/Makefile 文件中，定义了编译链接$(obj)/compressed/vmlinux 的规则。

```
#arch/arm/boot/compressed/Makefile
$(obj)/vmlinux: $(obj)/vmlinux.lds $(obj)/$(HEAD) $(obj)/piggy.o \
           $(addprefix $(obj)/, $(OBJS)) FORCE
     $(call if_changed,ld)
     @:
```

前提条件中，$(obj)/vmlinux.lds 是链接脚本，$(obj)/$(HEAD)是自引导的目标代码，$(obj)/piggy.o 是顶层 vmlinux 的精简压缩代码。为了保证自引导代码组装在映像起始位置，还要使用链接脚本。

3．编译内核模块

Linux 2.6 内核的模块采用新的加载器，它是由 Rusty Russel 开发的。它使用内核编译机制，生成一个*.ko（内核目标文件，kernel object）模块目标文件，而不是一个*.o 模块目标文件。

内核编译系统首先编译这些模块，然后链接上 vermagic.o。这样就在目标模块创建了一个特殊区域，用来记录编译器版本号、内核版本号、是否使用内核抢占等信息。

新的内核编译系统如何来编译并加载一个简单的模块的呢？举例一个简单的例子说明。我们写一个最简单的“hello”模块，只要实现模块初始化函数和退出函数就够了。这个模块程序叫作 hello.c。

```
#drivers/char/hello/hello.c
void init_module (void)
{
     printk( "Hello module!\n");
}
void cleanup_module (void);
{
     printk( "Bye module!\n");
}
```

相应的 Makefile 文件如下。

```
KERNEL_SRC = ~/linux-2.6.14
```

```
SUBDIR = $(KERNEL_SRC)/drivers/char/hello/
all: modules
obj-m := hello_mod.o
hello-objs := hello.o
EXTRA_FLAGS += -DDEBUG=1
modules:
    $(MAKE) -C $(KERNEL_SRC) SUBDIR=$(SUBDIR) modules
```

makefile 文件使用内核编译机制来编译模块。编译好的模块将被命名为 hello_mod.ko，它是编译 hello.c 并且链接 vermagic.o 而得到的。KERNEL_SRC 指定内核源文件所在的目录，SUBDIR 指定放置模块的目录。EXTRA_FLAGS 指定了需要给出的编译标记。

新模块要用新的模块工具加载或卸载。原来 2.4 内核的工具不能再用来加载或卸载 2.6 内核模块。2.4 的内核模块可能会发生使用和卸载冲突的情况，这是由于模块使用计数是由模块代码自己来控制的。新的模块加载工具可以尽量避免这种情况发生。

Linux 2.6 内核模块不再需要对引用计数进行加或减操作，这些工作已经由模块代码外部处理。任何要使用模块的代码都必须调用 try_module_get(&module)，只有在调用成功以后才能访问那个模块。如果被调用的模块已经被卸载，那么这次调用会失败。访问完成时，要通过 module_put()函数释放模块。

7.2.5 内核编译结果

相对于 Linux 2.4 内核，Linux 2.6 内核配置编译过程要简单一些，不再需要 make dep; make zImage; make modules 的命令。配置好内核之后，只要执行 make 就可以编译内核映像和模块。

内核的配置菜单选项内容也有了较大变化，我们在下一节中再详细讨论。

内核编译完成以后，将生成几个重要的文件。它们是 vmlinux、vmlinuz 和 System.map。

（1）vmlinux

vmlinux 是在内核源码顶层目录生成的内核映像。它是内核在虚拟空间运行时代码的真实反映。编译的过程就是按照特定顺序链接目标代码，生成 vmlinux。因为 Linux 内核运行在虚拟地址空间，所以名字附加“vm”（Virtual Memory）。Vmlinux 不具备引导的能力，需要借助其他 bootloader 引导启动。

（2）vmlinuz

vmlinuz 是可引导的、压缩的内核映像，也就是 zImage。它是 vmlinux 的压缩映像，是可执行的 Linux 内核映像。vmlinuz 的生成跟体系结构很有关系，不同体系结构的内核一般有不同的格式。大多数 vmlinuz 包含 2 部分：压缩的 vmlinux 和自引导程序。Vmlinuz 通过自引导程序初始化系统，并且解压启动 vmlinux。Vmlinuz 采用 gzip 压缩格式，都包含 gzip 的解压缩函数。

（3）System.map

System.map 是一个特定内核的内核符号表，它包含内核全局变量和函数的地址信息。

System.map 是内核编译生成文件之一。当 vmlinux 编译完成时，再通过$(NM)命令解析 vmlinux 映像生成。可以直接通过 nm 命令来查看任何一个可执行文件的信息。

```
$ nm vmlinux > System.map
```

不过，内核源码还要对 nm 生成的信息加以过滤排序，才能得到 System.map。

```
$ nm vmlinux | grep -v '\(compiled\)\|\(\.o$$\)\|\( [aUw] \)\|\(\.\.ng$$\)\|
\(LASH[RL]DI\)' | sort > System.map
```

Linux 内核是一个很复杂的代码块，有许许多多的全局符号。它不使用符号名，而是通过变量或函数的地址来识别变量或函数名。比如：不使用 size_t BytesRead 这样的符号，而使用地址 c0343f20 引用这个变量。

内核主要是用 C 写的，编译成目标代码或者映像就可以直接使用地址了。如果我们需要知道符号的地址，或者需要知道地址对应的符号，就需要由符号表来完成。符号表是所有符号连同它们的地址的列表。

System.map 是在内核编译过程中生成的，每一个内核映像对应自己的 System.map。它是保存在文件系统上的文件。当编译一个新内核时，各个符号名的地址要发生变化，就应当用新的 System.map 来取代旧的 System.map。它可以提供给 klogd、lsof 和 ps 等程序使用。

Linux 内核还有另外一种符号表使用方式：/proc/ksyms。它是一个“proc”接口，是在内核映像引导时创建的/proc/ksyms 条目。用户空间的程序可以通过/proc/ksyms 接口可以读取内核符号表。这需要预先配置 CONFIG_ALLKSYMS 选项，内核映像将包含符号表。

7.3 内核配置选项

基于内核配置系统，可以对内核的上千个选项进行配置。那么，这些选项应该如何使用呢？下面以 ARM 平台为例，介绍常用的内核配置选项。

7.3.1 使用配置菜单

内核配置过程比较繁琐，但是配置是否适当与 Linux 系统运行直接相关，所以需要了解一些主要选项的设置。

配置内核可以选择不同的配置界面，图形界面或者光标界面。由于光标菜单运行时不依赖于 X11 图形软件环境，可以运行在字符终端上，所以光标菜单界面比较通用。图 7.2 所示就是执行 make menuconfig 出现的配置菜单。

在各级子菜单项种，选择相应的配置时，有 3 种选择，它们代表的含义分别如下。

Y－将该功能编译进内核。

N－不将该功能编译进内核。

M－将该功能编译成可以在需要时动态插入到内核中的模块。

如果使用的是 make xconfig，使用鼠标就可以选择对应的选项。如果使用的是 make menuconfig，则需要使用回车键进行选取。

在每一个选项前都有个括号，有的是中括号，有的是尖括号，还有的是圆括号。用空格键选择时可以发现，中括号里要么是空，要么是“*”，而尖括号里可以是空，“*”和“M”。这表示前者对应的项要么不要，要么编译到内核里；后者则多一样选择，可以编译成模块。而圆括号的内容是要你在所提供的几个选项中选择一项。

在编译内核的过程中，最麻烦的事情就是这步配置工作了。初次接触 Linux 内核的开发者往往弄不清楚该如何选取这些选项。实际上在配置时，大部分选项可以使用其缺省值，只有小部分需要根据用户不同的需要选择。选择的原则是将与内核其他部分关系较远且不经常使用的部分功能代码编译成为可加载模块，有利于减小内核的长度，减小内核消耗的内存，简化该功能相应的环境改变时对内核的影响；不需要的功能就不要选；与内核关系紧密而且经常使用的部分功能代码直接编译到内核中。

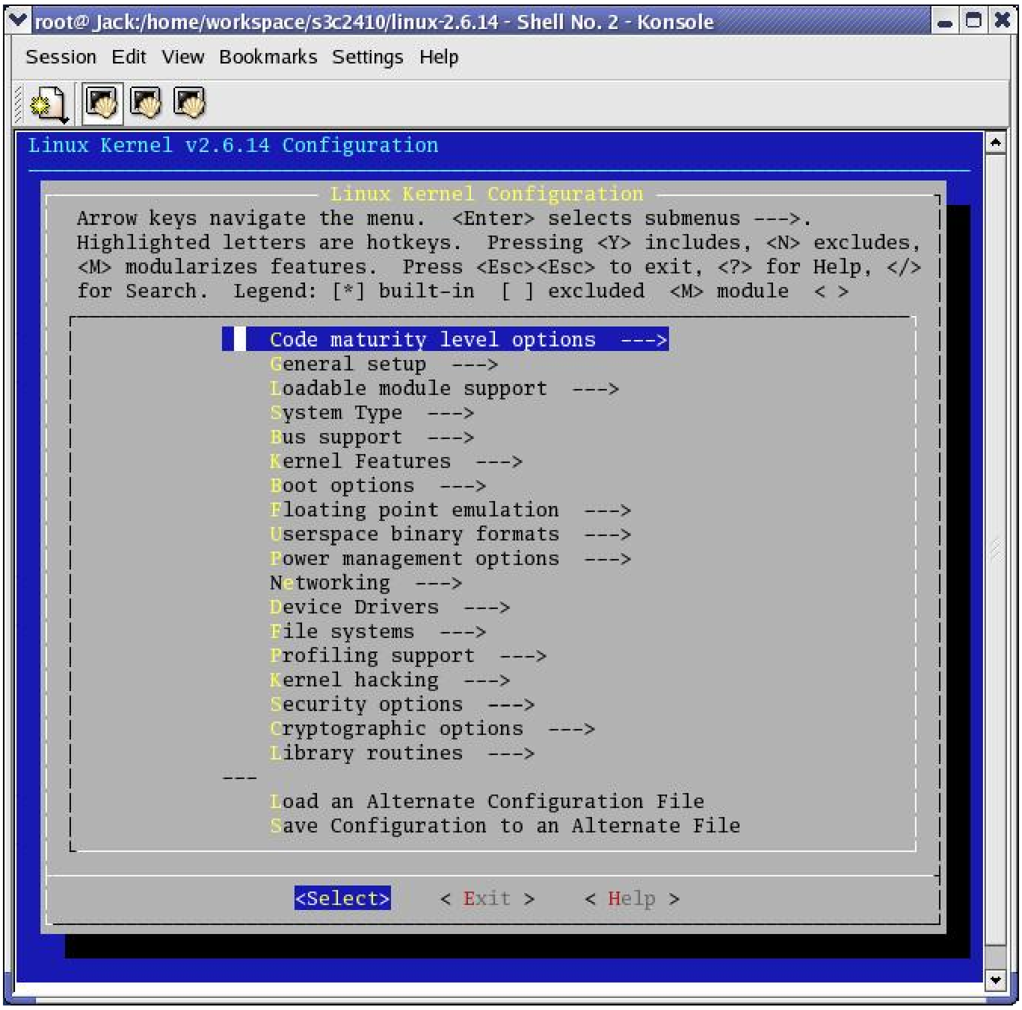


图 7.2　内核配置主菜单

7.3.2　基本配置选项

相对于 Linux 2.4 内核，2.6 内核的配置菜单有了很大变化，而且随着版本的发展还有些调整。下面以 Linux-2.6.14 内核版本为例，介绍主菜单选项和常用的配置选项的功能。

（1）“Code maturity level options”菜单包含配置控制代码成熟度的一些选项

CONFIG_EXPERIMENTAL 选项可以包含一些处于开发状态或者不成熟的代码或者驱动程序。

（2）“General setup”菜单包含通用的一些配置选项

CONFIG_LOCALVERSION 可以定义附加的内核版本号。

CONFIG_SWAP 可以支持内存页交换（swap）的功能。

CONFIG_EMBEDDED 支持嵌入式 Linux 标准内核配置。

CONFIG_KALLSYMS 支持加载调试信息或者符号解析功能。

（3）“Loadable module support”菜单包含支持动态加载模块的一些配置选项

CONFIG_MODULES 是支持动态加载模块功能选项。

CONFIG_MODVERSIONS 是模块版本控制支持选项。

CONFIG_KMOD 选项可以支持内核自动加载模块功能。

（4）“System Type”菜单包含系统平台列表及其相关的配置选项

对于不同的体系结构，显示不同的提示信息。ARM 体系结构显示“ARM system type”。

CONFIG_ARCH_CLPS7500 是 Cirrus Logic PS7500FE 开发板的配置选项。

还有其他很多处理器和板子的配置选项，不一一说明。

（5）“Bus support”菜单包含系统各种总线的配置选项

CONFIG_PCI 是 PCI 总线支持选项。

（6）“Kernel Features”菜单包含内核特性相关选项

CONFIG_PREEMPT 选项支持内核抢占特性。

CONFIG_SMP 选项支持对称多处理器的平台。

（7）“Boot options”菜单包含内核启动相关的选项

CONFIG_CMDLINE 选项可以定义缺省的内核命令行参数。

CONFIG_XIP_KERNEL 选项可以支持内核从 ROM 中运行的功能。

（8）“Floating point emulation”菜单包含浮点数运算仿真功能

CONFIG_FPE_NWFPE 选项支持“NWFPE”数学运算仿真。

CONFIG_FPE_FASTFPE 选项支持“FastFPE”数学运算仿真。

（9）“Userspace binary formats”菜单包含支持的应用程序格式

CONFIG_BINFMT_ELF 选项支持 ELF 格式可执行程序，这是 Linux 程序缺省的格式。

CONFIG_BINFMT_AOUT 选项支持 AOUT 格式可执行程序，现在已经少用。

（10）“Power management options”菜单包含电源管理有关的选项

CONFIG_PM 支持电源管理功能。

CONFIG_APM 支持高级电源管理仿真功能。

（11）“Networking”菜单包含网络协议支持选项

CONFIG_NET 选项支持网络功能。

CONFIG_PACKET 支持 socket 接口的功能。

CONFIG_INET 选项支持 TCP/IP 网络协议。

CONFIG_IPV6 选项支持 IPv6 协议的支持。

（12）"Device Drivers"菜单包含各种设备驱动程序

这个菜单下面包含很多子菜单，几乎包含了所有的设备驱动程序。我们将在第 7.3.3 节具体描述。

（13）"File systems"菜单包含各种文件系统的支持选项

CONFIG_EXT2_FS 选项支持 EXT2 文件系统。

CONFIG_EXT3_FS 选项支持 EXT3 文件系统。

CONFIG_JFS_FS 选项支持 JFS 文件系统。

CONFIG_INOTIFY 选项支持文件改变通知功能。

CONFIG_AUTOFS_FS 选项支持文件系统自动挂载功能。

"CD-ROM/DVD Filesystems"子菜单包含 iso9660 等 CD ROM 文件系统类型选项。

"DOS/FAT/NT Filesystems"子菜单包含 DOS/Windows 的一些文件系统类型选项。

"Pseudo filesystems"子菜单包含 sysfs procfs 等驻留在内存中的伪文件系统选项。

"Miscellaneous filesystems"子菜单包含 JFFS2 等其他类型的文件系统。

"Network File Systems"子菜单包含 NFS 等网络相关的文件系统。

（14）"Profiling support"菜单包含用于系统测试的工具选项

CONFIG_PROFILING 选项支持内核的代码测试功能。

CONFIG_OPROFILE 选项使能系统测试工具 Oprofile。

（15）"Kernel hacking"菜单包含各种内核调试的选项

这些选项的功能将在第 9.1.1 节详细介绍。

（16）"Security options"菜单包含安全性有关的选项

CONFIG_KEYS 选项支持密钥功能。

CONFIG_SECURITY 选项支持不同的密钥模型。

CONFIG_SECURITY_SELINUX 选项支持 NSA SELinux。

（17）"Cryptographic options"菜单包含加密算法

CONFIG_CRYPTO 选项支持加密的 API。

还有各种加密算法的选项可以选择。

（18）"Library routines"菜单包含几种压缩和校验库函数

CONFIG_CRC32 选项支持 CRC32 校验函数。

CONFIG_ZLIB_INFLATE 选项支持 zlib 压缩函数。

CONFIG_ZLIB_DEFLATE 选项支持 zlib 解压缩函数。

7.3.3 驱动程序配置选项

几乎所有 Linux 的设备驱动程序都在"Device Drivers"菜单下，它对设备驱动程序加以归类，放到子菜单下。下面解释常用的一些菜单项的内容。

（1）"Generic Driver Options"菜单对应 drivers/base 目录的配置选项，包含 Linux 驱动程

序基本和通用的一些配置选项。

（2）“Memory Technology Devices (MTD)”菜单对应 drivers/mtd 目录的配置选项，包含 MTD 设备驱动程序的配置选项。

（3）“Parallel port support”菜单对应 drivers/parport 目录的配置选项，包含并口设备驱动程序。

（4）“Plug and Play support”菜单对应 drivers/pnp 目录的配置选项，包含计算机外围设备的热拔插功能。

（5）“Block devices”菜单对应 drivers/block 目录的配置选项，包含软驱、RAMDISK 等驱动程序。

（6）“ATA/ATAPI/MFM/RLL support”菜单对应 drivers/ide 目录的配置选项，包含各类 ATA/ATAPI 接口设备驱动。

（7）“SCSI device support”菜单对应 drivers/scsi 目录的配置选项，包含各类 SCSI 接口的设备驱动。

（8）“Network device support”菜单对应 drivers/net 目录的配置选项，包含各类网络设备驱动程序。

（9）“Input device support”菜单对应 drivers/input 目录的配置选项，包含 USB 键盘鼠标等输入设备通用接口驱动。

（10）“Character devices”菜单对应 drivers/char 目录的配置选项，包含各种字符设备驱动程序。这个目录下的驱动程序很多。串口的配置选项也是从这个子菜单调用的，但是串口驱动所在的目录是 drivers/serial。

（11）“I^2C support”菜单对应 drivers/i2c 目录的配置选项，包含 I^2C 总线的驱动。

（12）“Multimedia devices”菜单对应 drivers/media 目录的配置选项，包含视频/音频接收和摄像头的驱动程序。

（13）“Graphics support”菜单对应 drivers/video 目录的配置选项，包含 Framebuffer 驱动程序。

（14）“Sound”菜单对应 sound 目录的配置选项，包含各种音频处理芯片 OSS 和 ALSA 驱动程序。

（15）“USB support”菜单对应 drivers/usb 目录的配置选项，包含 USB Host 和 Device 的驱动程序。

（16）“MMC/SD Card support”菜单对应 drivers/mmc 目录的配置选项，包含 MMC/SD 卡的驱动程序。

对于特定的目标板，可以根据外围设备选择对应的驱动程序选项，然后才能在 Linux 系统下使用相应的设备。

这里不准备讨论 Linux 设备驱动程序的话题。有关设备驱动程序的内容，可以阅读《Linux Device Drivers 3rd Edition》。

“黑色经典”系列之《嵌入式 Linux 系统开发技术详解——基于 ARM》

FARSIGHT

第 8 章 内核移植浅析

本章目标

本章以 ARM 平台为例介绍了内核移植的基本方法，并且详细分析了 Linux 内核启动过程。通过本章学习，可以明确内核哪些代码是与平台相关的，在内核启动过程中代码的执行顺序。只有掌握了这些代码，在内核移植过程中才能有的放矢地去修改代码。

- 内核源码移植
- Linux 内核启动过程分析

8.1 移植内核源码

所谓移植就是把程序代码从一种运行环境转移到另外一种运行环境。对于内核移植来说，主要是从一种硬件平台转移到另外一种硬件平台上运行。

8.1.1 移植前的准备工作

对于嵌入式 Linux 系统来说，有各种体系结构的处理器和硬件平台，并且用户需要根据需求自己定制硬件板。只要是硬件平台有些变化，即使非常小，可能也需要做一些移植工作。内核移植是嵌入式 Linux 系统中最常见的一项工作。

内核移植工作主要是修改跟硬件平台相关的代码，一般不涉及 Linux 内核通用的程序。移植的难度也取决于两种硬件平台的差异。Linux 对于特定的硬件平台的软件就叫作 BSP（Board Support Package）。

由于 Linux 内核具备可移植性的特点，并且已经支持了各种体系结构的很多种目标板，我们很容易从中找到跟自己硬件类似的目标板。参考内核已经支持的目标板来移植 BSP，就如同使用模板开发程序。

在开始移植开发板的 BSP 之前，需要做充分的准备工作。

（1）选择参考板

选择参考板的原则如下。

- 参考板与开发板具有相同的处理器，至少类似的处理器；
- 参考板和开发板具有相同的外围接口电路，至少基本接口相同；
- Linux 内核已经支持参考板，至少有非官方的补丁或者 BSP；
- 参考板 Linux 设备驱动工作正常，至少已经驱动基本接口。

通常都可以找到相同处理器的参考板，并且可以获取到 Linux 内核源代码。因为半导体商在发布一块新的处理器的时候，一般会为它提供参考设计板和 Linux BSP。即使是一款新的处理器，也可以找到体系结构相同、功能相似的处理器作为参考。

还要仔细分析内核代码，弄清楚哪些设备有驱动程序，哪些还没有。如果某个驱动程序还没有支持，就需要我们自己动手写驱动了。

（2）编译测试参考板的 Linux 内核

为了确信 Linux 对参考板的支持情况，最好验证一下。配置编译 Linux 内核，在目标板上运行测试一下。

也许最新的 Linux 内核版本支持的最好，但是也可能需要在老内核版本上打补丁。可以都测试一下，总之要选择硬件平台支持最好、版本最新的内核。

对于交叉开发来说，首先要在顶层 Makefile 中设置 ARCH、CROSS_COMPILE 和 EXTRA_VERSION 变量，然后才能选择配置指定的体系结构平台。ARM 平台的例子如下。

```
ARCH    := arm
CROSS_COMPILE   :=  arm-linux-
EXTRA_VERSION   :=
```

可以使用参考板的缺省内核配置，这可以在 arch/$(ARCH)/configs 目录下找到。以 smdk2401 为例，arch/arm/configs/smdk2410_defconfig 就是缺省文件。执行下列命令使缺省配置生效。

```
$ make smdk2410_defconfig
```

再可以打开配置菜单，重新调整配置选项。

确认保存配置以后，执行 make 编译内核。

编译完成后，得到的内核映像文件是 arch/arm/boot/zImage。

内核模块都可以安装到目标板文件系统中，以便加载测试。例如：目标板根文件系统目录是<dir>，执行下列命令后，模块都安装到<dir>/lib/modules/目录树中。

```
$ make INSTALL_MOD_PATH=<dir>  modules_install
```

（3）分析参考板的 BSP 代码

如果 Linux 内核基本支持这个参考板，那么还要进一步熟悉这部分程序，因为它将是下一步移植的基础和模板。

分析平台相关的部分代码实现；分析内核编译组织方式；分析内核启动的初始化程序；分析驱动程序的实现。第 8.1.2 节详细讨论有关的代码。

熟悉了参考板的这些代码以后，就可以动手修改内核源代码了。

8.1.2　开发板内核移植

对于内核移植工作来说，主要是添加开发板初始化和驱动程序的代码。这部分代码大部分是跟体系结构相关的，在 arch 目录下按照不同的体系结构管理。下面以 ARM S3C2410 平台为例，分析内核代码移植过程。

Linux 2.6 内核已经支持 S3C2410 处理器的多种硬件板，例如：SMDK2410、Simtec-BAST、IPAQ-H1940、Thorcom-VR1000 等。我们可以参考 SMDK2410 参考板，来移植开发板的内核。

1．添加开发板平台支持选项

Linux 2.6.14 内核对 S3C2410 平台已经有基本的支持。这部分可以省略了，不过从学习的角度，再分析一下 ARM S3C2410 平台的有关代码实现。

回顾一下第 7.3 节中内核配置选项的“System Type”，其中有处理器以及开发板的支持选项。那么它们是怎么加进去的呢？又起什么作用呢？

这些 ARM 平台相关的选项都是在 arch/arm 目录下实现的。在内核编译过程中已经说明，需要在顶层 Makefile 中设置相应的体系结构和工具链。这样配置 Linux 内核的时候就会调用 arch/arm/Kconfig 文件。

arch/arm/Kconfig 文件是内核主配置文件，从这个文件中就可以找到“System Type”的配置选项。

```
#arch/arm/Kconfig
menu "System Type"
choice      #系统平台选择项列表
     prompt "ARM system type"
```

```
        default ARCH_RPC
config ARCH_CLPS7500
        bool "Cirrus-CL-PS7500FE"
        select TIMER_ACORN
        select ISA
……
config ARCH_S3C2410     #对于 S3C2410 处理器的支持
        bool "Samsung S3C2410"
        help
             Samsung S3C2410X CPU based systems, such as the Simtec Electronics
             BAST (<http://www.simtec.co.uk/products/EB110ITX/>), the IPAQ 1940 or
             the Samsung SMDK2410 development board (and derviatives).
……
config ARCH_AAEC2000
        bool "Agilent AAEC-2000 based"
        help
             This enables support for systems based on the Agilent AAEC-2000
endchoice
……
source "arch/arm/mach-s3c2410/Kconfig"
```

上面的“choice”语句可以在菜单中生成一个多选项，可以找到“Samsung S3C2410”选项，然后通过 source 语句调用 arch/arm/mach-s3c2410/Kconfig 文件。

arch/arm/mach-s3c2410/Kconfig 文件中定义了各种 S3C2410 处理器开发板的选项，还有 S3C2410 处理器的特殊支持选项。

```
 # arch/arm/mach-s3c2410/Kconfig
if ARCH_S3C2410
menu "S3C24XX Implementations"      #S3C24XX 系列开发板的选项
config MACH_ANUBIS
        bool "Simtec Electronics ANUBIS"
        select CPU_S3C2440
        help
            Say Y gere if you are using the Simtec Electronics ANUBIS
            development system
……
config ARCH_SMDK2410          #SMDK2410 开发板的配置选项
        bool "SMDK2410/A9M2410"
        select CPU_S3C2410
        help
```

```
        Say Y here if you are using the SMDK2410 or the derived module A9M2410
           <http://www.fsforth.de>
……
endmenu

config CPU_S3C2410      #根据依赖关系缺省定义处理器选项
     bool
     depends on ARCH_S3C2410
     help
        Support for S3C2410 and S3C2410A family from the S3C24XX line
        of Samsung Mobile CPUs.

comment "S3C2410 Boot"      #内核启动阶段使能看门狗的选项
config S3C2410_BOOT_WATCHDOG
     bool "S3C2410 Initialisation watchdog"
     depends on ARCH_S3C2410 && S3C2410_WATCHDOG
     help
        Say y to enable the watchdog during the kernel decompression
        stage. If the kernel fails to uncompress, then the watchdog
        will trigger a reset and the system should restart.
        Although this uses the same hardware unit as the kernel watchdog
        driver, it is not a replacement for it. If you use this option,
        you will have to use the watchdg driver to either stop the timeout
       or restart it. If you do not, then your kernel will reboot after
        startup.
        The driver uses a fixed timeout value, so the exact time till the
        system resets depends on the value of PCLK. The timeout on an
        200MHz s3c2410 should be about 30 seconds.

comment "S3C2410 Setup"     #S3C2410 处理器有关的配置选项，例如：DMA 的支持等
config S3C2410_DMA
     bool "S3C2410 DMA support"
     depends on ARCH_S3C2410
     help
        S3C2410 DMA support. This is needed for drivers like sound which
        use the S3C2410's DMA system to move data to and from the
        peripheral blocks.
……
endif
```

通过上述的两个 Kconfig 文件，提供了处理器和目标板以及处理器特性的选择项。当选择了“S3C24XX Implementations”的时候，自动出现 S3C2410 系列开发板的配置菜单选项。

这里的 mach-s3c2410 目录专门用来保存 S3C2410 系列处理器平台相关程序。下面列出 mach-s3c2410 目录下的所有文件。

```
$ ls arch/arm/mach-s3c2410
bast.h          irq.c              mach-smdk2410.c    s3c2440.c
bast-irq.c      irq.h              mach-smdk2440.c    s3c2440.h
clock.c         Kconfig            mach-vr1000.c      s3c2440-clock.c
clock.h         mach-anubis.c      Makefile           s3c2440-dsc.c
cpu.c           mach-bast.c        Makefile.boot      s3c2440-irq.c
cpu.h           mach-h1940.c       pm.c               sleep.S
devs.c          mach-n30.c         pm.h               time.c
devs.h          mach-nexcoder.c    pm-simtec.c        usb-simtec.c
dma.c           mach-otom.c        s3c2410.c          usb-simtec.h
gpio.c          mach-rx3715.c      s3c2410.h
```

其中 Kconfig 和 Makefile 是用于内核配置编译的。其他文件分为 2 类，一类是处理器通用的，例如：clock.c clock.h cpu.c cpu.h s3c2410.c s3c2410.h 等；另一类是目标板相关的，例如：bast.h bast-irq.c mach-bast.c 等。

在这些文件中，实现了处理器和目标板相关的一些定义和初始化函数。还有些相关的定义包含在 include/asm-arm/arch-s3c2410/下的头文件中。

那么看一下 SMDK2410 目标板在内核中是如何描述的。

先来看一下 MACHINE_START 和 MACHINE_END 宏的定义。

```
/* include/asm-arm/mach/arch.h */
#define MACHINE_START(_type,_name)              \
const struct machine_desc __mach_desc_##_type  \
__attribute__((__section__(".arch.info.init"))) = {    \
     .nr    = MACH_TYPE_##_type,    \
     .name  = _name,
#define MACHINE_END                    \
};
```

其中的结构体 machine_desc 用来描述目标板硬件平台。它包含了系统平台号（nr，architecture number）、内存起始物理地址（phys_ram）、I/O 起始物理地址（phys_io）、系统平台名称（name）、启动参数（boot_params）以及初始化函数指针等变量。

再来定义 SMDK2410 这个系统平台。

```
#arch/arm/mach-s3c2410/mach-smdk2410.c
MACHINE_START(SMDK2410, "SMDK2410") /* 定义 SMDK2410 的结构体 */
    /* Maintainer: Jonas Dietsche */
```

```
    .phys_ram    = S3C2410_SDRAM_PA,
    .phys_io     = S3C2410_PA_UART,
    .io_pg_offst= (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
    .boot_params= S3C2410_SDRAM_PA + 0x100,
    .map_io      = smdk2410_map_io,
    .init_irq    = smdk2410_init_irq,
    .timer       = &s3c24xx_timer,
MACHINE_END
```

上面相当于定义了下列__mach_desc_SMDK2410 结构体。

```
const struct machine_desc __mach_desc_SMDK2410 \
 __attribute__((__section__(".arch.info.init"))) = {   \
     .nr         = MACH_TYPE_SMDK2410,   \
     .name       = "SMDK2410",
     ……
};
```

这里的 MACH_TYPE_SMDK2410 是 SMDK2410 的系统平台号，它包含在 include/asm-arm/mach-types.h 头文件中。不过这个头文件是自动生成的，不能手工修改。真正系统平台号的定义位置在 arch/arm/tools/mach-types 文件中。

```
#arch/arm/tools/mach-types
# machine_is_xxx        CONFIG_xxxx          MACH_TYPE_xxx    number
smdk2410                ARCH_SMDK2410        SMDK2410         193
```

arch/arm/tools/mach-types 中每一行定义一个系统平台号。“machine_is_xxx”是用来判断当前的平台号是否正确的函数；“CONFIG_xxxx”是在内核配置时生成的；“MACH_TYPE_xxx”是系统平台号的定义；“number”是系统平台的值。

在_mach_desc_SMDK2410 结构体中，还有一些系统平台初始化函数，例如：smdk2410_map_io()、smdk2410_init_irq()、s3c24xx_timer()等。这些函数分别在其他文件中逐一实现。在内核启动过程中，将通过结构体调用这些函数，完成系统平台初始化工作。

内核中已经支持各种系统平台，例如：mach-clps711x、mach-integrator、mach-omap1 等。在 Makefile 中可以通过配置来选择编译不同的目录，arch/arm/Makefile 的下列语句可以完成这项工作。

```
#arch/arm/Makefile
machine-$(CONFIG_ARCH_S3C2410)    := s3c2410    #定义 machine-y = s3c2410
……
ifneq ($(machine-y),)
MACHINE  := arch/arm/mach-$(machine-y)/     #包含 mach-s3c2410 子目录
else
MACHINE  :=
Endif
```

然后应该可以编译内核映像了，内核的编译过程可以参考第 7.2 节的内容。

编译生成顶层的 vmlinux 映像之后，还需要把它压缩打包成自引导的内核映像 zImage。这部分代码都放在 arch/arm/boot/目录下。

自引导代码主要包含在 arch/arm/boot/compressed/head.S 文件中，这个文件将在第 8.2 节分析。这里对 zImage 的编译生成过程简单分析一下。

```
# arch/arm/boot/compressed/Makefile
$(obj)/vmlinux: $(obj)/vmlinux.lds $(obj)/$(HEAD) $(obj)/piggy.o \
           $(addprefix $(obj)/, $(OBJS)) FORCE
      $(call if_changed,ld)
      @:
$(obj)/piggy.gz: $(obj)/../Image FORCE
      $(call if_changed,gzip)
$(obj)/piggy.o:  $(obj)/piggy.gz FORCE

$(obj)/vmlinux.lds: $(obj)/vmlinux.lds.in arch/arm/boot/Makefile .config
       @sed "$(SEDFLAGS)" < $< > $@
$(obj)/misc.o: $(obj)/misc.c include/asm/arch/uncompress.h lib/inflate.c
```

根据上述 Makefile 的定义，先把顶层的 vmlinux 转换成 Image，再压缩成 gzip 格式的 piggy.gz，再生成 piggy.o，然后链接生成新的 vmlinux。这里的 vmlinux 将复制成 zImage，它是在链接时需要一个链接脚本 vmlinux.lds。这里的链接脚本相对简单多了，只要保证自引导程序组装在 zImage 的起始位置即可。

幸运的是 Linux 2.6 内核已经支持 S3C2410 处理器，这部分体系结构相关的程序基本上都有了。如果社区没有支持你的硬件平台，模仿这种源代码的组织结构，动手移植吧。

在移植过程中，内核编译也可能出现一些错误。最常见的配置错误，例如：找不到头文件或者宏定义，在目标文件编译过程中就会出错；找不到函数实现，在链接的时候出错。

还有一些语法错误，通常因为编辑失误导致，另外不同内核版本的函数接口定义不一致也会导致。根据错误信息，很容易就可以找到出错的位置。

开始移植的时候，可以先配置一个最基本的 Linux 内核，甚至不包含串口驱动、网络驱动。

2．移植开发板驱动程序

S3C2410 属于片上系统，处理器芯片具备串口、显示等外围接口的控制器。这样，参考板上的设备驱动程序多数可以直接使用。但是并不是所有的外部设备都相同，不同的开发板可以使用不同的 SDRAM、Flash、以太网接口芯片等。这就需要根据硬件修改或者开发驱动程序。

串口驱动程序是最简单的设备驱动程序之一，这个驱动程序几乎不需要任何改动。然而，如果用 2.4 内核的配置使用方式，是不能得到串口控制台信息的。看一下驱动程序 drivers/serial/s3c2410.c 中的一些代码就明白了。

```
/* drivers/serial/s3c2410.c */
/* UART name and device definitions */
#define S3C24XX_SERIAL_NAME "ttySAC"/* 设备名称由 2.4 内核的 ttyS 变为 ttySAC */
#define S3C24XX_SERIAL_DEVFS    "tts/"
#define S3C24XX_SERIAL_MAJOR    204
#define S3C24XX_SERIAL_MINOR    64

static struct uart_driver s3c24xx_uart_drv = {
        .owner        = THIS_MODULE,
        .dev_name     = "s3c2410_serial",
        .nr       = 3,
        .cons         = S3C24XX_SERIAL_CONSOLE,
        .driver_name = S3C24XX_SERIAL_NAME,
        .devfs_name  = S3C24XX_SERIAL_DEVFS,
        .major        = S3C24XX_SERIAL_MAJOR,
        .minor        = S3C24XX_SERIAL_MINOR,
};
```

这样，串口设备在/dev 目录下对应的设备节点为：/dev/ttySAC0、/dev/ttySAC1。所以，再使用过去的串口设备 ttyS0，就得不到控制台打印信息了。

现在可以很简单地解决这个问题，把内核命令行参数的控制台设置修改为：console=ttySAC0，115200。

SMDK2410 开发板的网络接口使用 CS8900A 芯片，不过在 Linux 2.6.14 的内核中还没有支持。其实，Linux 社区已经有了针对这个网络接口的补丁。

网络驱动是比较复杂的驱动之一，这里不详细讲述驱动程序编程。我们重点了解一下移植过程到底做了哪些工作。这个 CS8900 10M 以太网接口驱动程序是 drivers/net/cs89x0.c。

```
/* drivers/net/cs89x0.c */
#ifdef CONFIG_ARCH_SMDK2410
#include <asm/irq.h>
#include <asm/hardware.h>
#undef inw
#define inw(p)      readw(p)
#undef insw
#define insw(p,d,l) readsw((void *) p, d, l)
#undef outw
#define outw(v, p)  writew(v, p)
#endif
```

对于特定的设备驱动，必须定义设备底层操作函数，也就是寄存器访问函数。外围设备的访问分为内存映射和 I/O 两种类型，分别根据各自体系结构实现这些函数。对于 SMDK2410

平台，是内存映射型的读写函数，通过宏定义调用 read/write 函数。

对于 SMDK2410 硬件平台，CS8900 以太网控制器的基地址和中断等配置与其他平台是不同的。通过下列程序可以为 SMDK2410 定义初始化数据。

```
#elif defined(CONFIG_ARCH_SMDK2410)
static unsigned int netcard_portlist[] __initdata = {SMDK2410_ETH_BASE + 0x300, 0};
static unsigned int cs8900_irq_map[] = {SMDK2410_ETH_IRQ, 0, 0, 0};
```

上面程序中用到的 SMDK2410_ETH_BASE 和 SMDK2410_ETH_IRQ，可以在头文件中添加。这是完全根据 SMDK2410 硬件使用手册或者电路图来确定的。

```
/* include/asm-arm/arch-s3c2410/smdk2410.h */
/*
 * include/asm-arm/arch-s3c2410/smdk2410.h
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */
#ifndef __ASM_ARCH_SMDK2410_H
#define __ASM_ARCH_SMDK2410_H
#include <linux/config.h>
#define SMDK2410_ETH_BASE   0xE9000000
#define SMDK2410_ETH_START  0x19000000
#define SMDK2410_ETH_IRQ    IRQ_EINT9
#endif  /* __ASM_ARCH_SMDK2410_H */
```

网络驱动程序修改好了，可能还不能找到这个网卡的驱动选项。这是因为 cs89x0 驱动依赖于其他配置选项。按照下列代码在“depends”一行添加“|| ARCH_SMDK2410”。

```
# drivers/net/Kconfig
config CS89x0
     tristate "CS89x0 support"
     depends on (NET_PCI && (ISA || ARCH_IXDP2X01)) || ARCH_PNX0105 || ARCH_SMDK2410
Makefile 的下列一行可以编译 CS89x0 驱动。
# drivers/net/Makefile
obj-$(CONFIG_CS89x0) += cs89x0.o
```

如果有更多的设备接口，可以参考 drivers 目录中各种成熟的设备驱动。这里不再详细讨论设备驱动程序的内容。

8.1.3 移植后的工作

在内核移植过程中，最好通过版本控制工具来维护内核源代码，至少多做一些备份。因为手工修改代码比较麻烦，但是删除却很容易。

移植完成以后，就可以发布这个内核源代码了。最常见的方式是发布内核补丁。基于一个稳定的内核版本制作补丁文件，可以方便地保存和分发。

假设基于 linux-2.6.14 内核移植，没有修改的内核源代码目录是 linux-2.6.14，修改过的内核源代码目录是 linux-2.6.14-smdk2410。按照下列步骤制作补丁。

```
$ cd linux-2.6.14-smdk2410
$ cp .config arch/arm/configs/smdk2410_defconfig
$ make mrproper
$ cd ../
$ diff -aur ./linux-2.6.14 ./linux-2.6.14-smdk2410 > patch-linux-2.6.14-smdk2410
```

这样就得到了一个补丁文件 patch-linux-2.6.14-smdk2410，还可以把它压缩保存。

```
$ gzip patch-linux-2.6.14-smdk2410
```

以后可以直接阅读这个补丁文件，或者通过 patch 工具打补丁。补丁工具的使用方法参考第 8.2 节。

8.2 Linux 内核启动过程分析

Linux 内核启动就是引导内核映像启动的过程。典型的内核映像是 zImage，包含自引导程序和压缩的 vmlinux 2 部分。启动过程也就是解压和启动 vmlinux 的过程。本章的与体系结构相关的代码都以 ARM 平台为例分析。

8.2.1 内核启动流程源代码分析

启动过程从内核映像入口开始执行，解压 vmlinux 并且转换到虚拟地址空间，再调用统一的内核启动函数 start_kernel()，从而启动整个 Linux 系统。

从内核源代码的角度分析，启动过程是一系列汇编子程序和 C 函数的调用过程。内核启动过程函数调用顺序如图 8.1 所示。

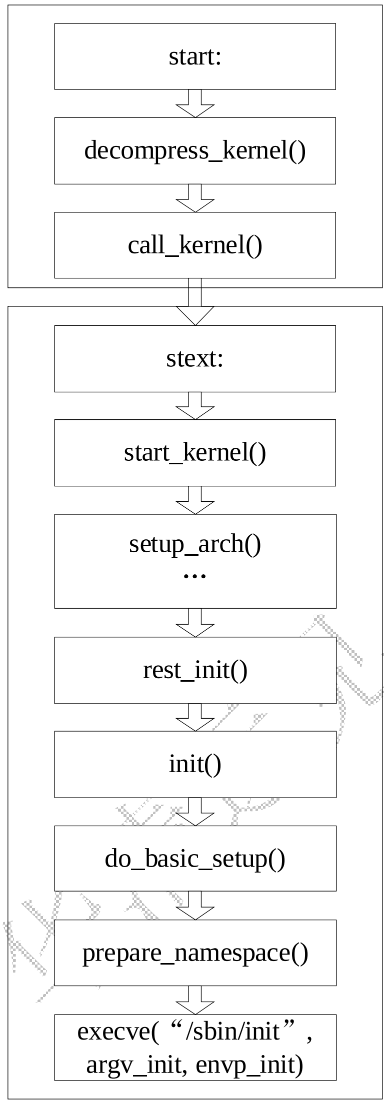


图 8.1　内核启动流程图

8.2.2　内核自引导程序

zImage 映像的入口代码是自引导程序。自引导程序包含一些初始化代码，所以它是体系结构相关的，这个目录是 arch/$(ARCH)/boot。那么第一条指令所在的文件是自引导程序中的 head.S。分析一下这部分汇编程序，就能清楚内核引导的过程。

```
/* arch/arm/boot/compressed/head.S */
       .align
start:
       .type    start,#function
       .rept    8
```

```
        mov  r0, r0
        .endr

        b1f
        .word    0x016f2818  @ 辅助引导程序的幻数
        .word    start       @ 加载运行 zImage 的绝对地址
        .word    _edata      @ zImage 结尾地址
1:      mov  r7, r1          @ 保存体系结构 ID
        mov  r8, #0          @ 保存 r0 寄存器

        teqp pc, #0x0c000003    @ 关闭中断

        /* 注意这里可能需要做 cache 刷新和其他工作 */
        /* 链接的时候，这里可以插入一些体系结构相关的代码，但是应该保留 r7 r8 */

        .text
        adr  r0, LC0
        ldmia    r0, {r1, r2, r3, r4, r5, r6, ip, sp}
        subs r0, r0, r1      @ 计算偏移量
        @ 如果偏移量为零，说明程序是运行在所链接的地址
        beq  not_relocated


        /* 偏移量不为零，说明运行在不同的地址，那么需要修正几个指针
         *   r5 – zImage 基地址
         *   r6 – GOT（全局偏移表）起始地址
         *   ip – GOT 结束地址
         */
        add  r5, r5, r0
        add  r6, r6, r0
        add  ip, ip, r0

#ifndef CONFIG_ZBOOT_ROM
          /* CONFIG_ZBOOT_ROM = n，完全运行在 PIC 状态;
           * 这时需要修正 BSS 区域的指针，S3C2410 平台适用。
           *   r2 – BSS 起始地址
           *   r3 – BSS 结束地址
           *   sp – 堆栈指针
           */
```

```
        add r2, r2, r0
        add r3, r3, r0
        add sp, sp, r0
        /* 重新定位 GOT 表中的所有项 */
1:      ldr r1, [r6, #0]        @ relocate entries in the GOT
        add r1, r1, r0          @ table.  This fixes up the
        str r1, [r6], #4        @ C references.
        cmp r6, ip
        blo 1b
#else  /* CONFIG_ZBOOT_ROM = y */
          /* 重新定位 GOT 表中的项
           * 只要重新定位 BSS 区域（已重定位）之外的部分
           */
1:        ldr    r1, [r6, #0]   @ relocate entries in the GOT
          cmp    r1, r2         @ entry < bss_start ||
          cmphs  r3, r1         @ _end < entry
          addlo  r1, r1, r0     @ table.  This fixes up the
          str    r1, [r6], #4   @ C references.
          cmp    r6, ip
          blo    1b
#endif
/*不需要重定位或者已经重定位过*/
not_relocated:  mov r0, #0
1:         str   r0, [r2], #4        @ 清除 bss
          str    r0, [r2], #4
          str    r0, [r2], #4
          str    r0, [r2], #4
          cmp    r2, r3
          blo    1b
          /* 这时 C 运行环境应该已经配置好了。
           *打开 cache，设置一些指针，开始解压 vmlinux
           */
          bl     cache_on
          mov    r1, sp            @ 在栈上面分配空间
          add    r2, sp, #0x10000 @ 最大 64KB

/* 检查是否会覆盖内核映像本身
 *   r4 = 内核结束地址
 *   r5 = 本映像的起始地址
```

```
 *   r2 = 分配空间的结束地址（并且处于本映像的前面）
 * 基本要求: r4 >= r2 或者 r4 + 映像长度 <= r5
 */
         cmp r4, r2
         bhs wont_overwrite  @ 如果 r4 >= r2，跳转到 wont_overwrite
         add r0, r4, #4096*1024  @ 内核最大 4MB
         cmp r0, r5
         bls wont_overwrite  @ 如果 r4 + 4MB <= r5，跳转到 wont_overwrite
/* 不满足上述 2 个条件的情况下 */
         mov r5, r2          @ 解压程序从分配空间后面存放
         mov r0, r5
         mov r3, r7
         bl  decompress_kernel
         add r0, r0, #127
         bic r0, r0, #127        @ 对齐内核长度
/*
 * r0     = 解压后内核长度
 * r1-r3  = 未使用
 * r4     = 内核执行地址
 * r5     = 解压内核起始地址
 * r6     = 处理器 ID
 * r7     = 体系结构 ID
 * r8-r14 = 未使用
 */
           add   r1, r5, r0      @ 解压后内核的结束地址
           adr   r2, reloc_start
           ldr   r3, LC1
           add   r3, r2, r3
1:         ldmia r2!, {r8 - r13}     @ 复制重定位代码
           stmia r1!, {r8 - r13}
           ldmia r2!, {r8 - r13}
           stmia r1!, {r8 - r13}
           cmp   r2, r3
           blo   1b
           bl    cache_clean_flush
           add   pc, r5, r0      @ 调用重定位代码

/* 这里不存在覆盖自身的问题了
 * r4     = 内核执行地址
```

```
 * r7     = 体系结构 ID
 */
wont_overwrite: mov r0, r4
          mov   r3, r7
          bl    decompress_kernel   @调用解压内核映像的 C 函数
          b call_kernel             @调用跳转到内核映像入口的子程序
/* call_kernel 子程序在启动过程中非常关键，负责跳转到内核映像的入口。 */
call_kernel:    bl  cache_clean_flush
          bl    cache_off
          mov   r0, #0
          mov   r1, r7      @ 保存体系结构号
          mov   pc, r4      @ 调用内核
```

其中 decompress_kernel 和 call_kernel 是最重要的 2 个子程序。

call_kernel 子程序完成启动 vmlinux 的任务，它负责关闭 CACHE，在 R2 寄存器中恢复系统平台号，然后跳转到 vmlinux 的入口。这样控制权就完全交给 vmlinux 执行了。

decompress_kernel 函数则是 C 语言写的，在 misc.c 文件中实现。

```
/* arch/arm/boot/compressed/misc.c */
ulg
decompress_kernel(ulg output_start, ulg free_mem_ptr_p, ulg free_mem_ptr_end_p,
           int arch_id)
{
    output_data         = (uch *)output_start;  /* 指向内核起始地址 */
    free_mem_ptr        = free_mem_ptr_p;
    free_mem_ptr_end    = free_mem_ptr_end_p;
    __machine_arch_type= arch_id;
    arch_decomp_setup();    /* 解压缩前的初始化和设置，包括串口波特率设置等 */
    makecrc();      /* CRC 校验 */
    putstr("Uncompressing Linux...");  /* 打印信息说明正在解压 Linux…. */
    gunzip();       /* 解压缩函数 */
    putstr(" done, booting the kernel.\n");    /* 解压完成 */
    return output_ptr;
}
```

8.2.3 内核 vmlinux 入口

PC 指针已经指向 vmlinux 的入口地址，顺序执行内核启动程序。vmlinux 开始部分也有一些汇编程序，对应的程序文件也叫作 head.S。

```
/* arch/arm/kernel/head.S */
```

```
/* 内核启动入口点
 * Kernel startup entry point.
 * 这里通常在解压后直接调用。
 * 处理器基本状态要求:
 * MMU 关闭, D-cach 关闭, I-cache 不用关系;
 * r0 = 0, r1 = 系统号 (machine number)
 * 这段代码几乎是位置无关的。
 * 如果链接内核在 0xc0008000, 调用的地址为相应的物理地址__pa(0xc0008000)。
 * r1 的系统号参考 arch/arm/tools/mach-types 文件的列表。
 * 尽量不要在这里添加系统号相关的代码, 那应该放在 bootloader 的代码中。
 * 保持这里的代码的整洁。
 */
    __INIT
    .type   stext, %function
ENTRY(stext)
    msr cpsr_c, #PSR_F_BIT | PSR_I_BIT | MODE_SVC @ 确定 SVC 模式并且关闭 IRQ
    bl  __lookup_processor_type@ r5=procinfo r9=cpuid
    movs    r10, r5             @ 若 r5==0, 则处理器类型不正确
    beq __error_p               @ 调用报错子程序打印"Error:p"
    bl  __lookup_machine_type   @ r5=machinfo
    movs    r8, r5              @若 r5==0, 则系统号不正确
    beq __error_a               @调用报错子程序打印"Error:a"
    bl  __create_page_tables

    /* 下列以位置无关的方式调用了 CPU 相关的代码。参考 arch/arm/mm/proc-*.S。
     * r10 是__lookup_machine_type 选择的 xxx_proc_info 结构体的基地址.
     * 返回时, CPU 就准备打开 MMU, r0 是 CPU 控制寄存器的值。
     */
    ldr r13, __switch_data      @ MMU 打开时要跳转到的地址
    adr lr, __enable_mmu        @ 返回 (PIC) 地址
    add pc, r10, #PROCINFO_INITFUNC

    .type   __switch_data, %object
__switch_data:
    .long   __mmap_switched
    .long   __data_loc          @ r4
    .long   __data_start        @ r5
    .long   __bss_start         @ r6
    .long   _end                @ r7
```

```
        .long   processor_id        @ r4
        .long   __machine_arch_type @ r5
        .long   cr_alignment        @ r6
        .long   init_thread_union + THREAD_START_SP @ sp

/* 下列代码段在 MMU 打开的情况下执行，使用绝对地址，代码不再位置无关。
 *  r0  = cp#15 control register
 *  r1  = machine ID
 *  r9  = processor ID
 */
        .type   __mmap_switched, %function
__mmap_switched:
        adr r3, __switch_data + 4

        ldmia   r3!, {r4, r5, r6, r7}
        cmp r4, r5              @ 如果需要，则复制代码段
1:  cmpne   r5, r6
        ldrne   fp, [r4], #4
        strne   fp, [r5], #4
        bne 1b

        mov fp, #0              @清除 BSS 并且 fp 置零
1:  cmp r6, r7
        strcc   fp, [r6],#4
        bcc 1b

        ldmia   r3, {r4, r5, r6, sp}
        str r9, [r4]            @ 保存处理器 ID
        str r1, [r5]            @ 保存系统类型
        bic r4, r0, #CR_A       @ 清除位"A"
        stmia   r6, {r0, r4}    @ 保存控制寄存器的值
        b   start_kernel
```

经过一系列的初始化过程，打开 MMU，跳转到 start_kernel()函数。

8.2.4 Linux 系统初始化

start_kernel 函数是 Linux 内核通用的初始化函数。无论对于什么体系结构的 Linux，都要执行这个函数。start_kernel()是内核初始化的基本过程。

```
/* init/main.c */
```

```
/* 激活第一个处理器 */
asmlinkage void __init start_kernel(void)
{
     char * command_line;
     extern struct kernel_param __start___param[], __stop___param[];
/* 这是仍然关闭中断。作必要的设置，然后再开中断。 */
     lock_kernel();
     page_address_init();
     printk(KERN_NOTICE);
     printk(linux_banner);
     setup_arch(&command_line);
     setup_per_cpu_areas();

     /* 标记启动 CPU"online"，以便让它在 printk 函数中调用控制台驱动，
      * 并且访问每一个 CPU 的自己的缓冲区。
      */
     smp_prepare_boot_cpu();
     /* 在启动任何中断（例如定时器中断）之前，设置调度器。
      * 完整的拓扑配置在 smp_init()处完成。但是我们需要一个功能调度。
      */
     sched_init();
     /* 关闭抢占，因为早期的启动调度是非常脆弱的。 */
     preempt_disable();
     build_all_zonelists();
     page_alloc_init();
     printk(KERN_NOTICE "Kernel command line: %s\n", saved_command_line);
     parse_early_param();
     parse_args("Booting kernel", command_line, __start___param,
               __stop___param - __start___param,
               &unknown_bootoption);
     sort_main_extable();
     trap_init();
     rcu_init();
     init_IRQ();
     pidhash_init();
     init_timers();
     softirq_init();
     time_init();
```

```
        /* 注意：这里是启动过程较早的阶段。
         * 控制台将比 PCI 等设备提前配置打开，console_init 函数一定要知道。
         * 我们希望早点打印，能够看到出错信息。
         */
        console_init();
        if (panic_later)
              panic(panic_later, panic_param);
        profile_init();
        local_irq_enable();
#ifdef CONFIG_BLK_DEV_INITRD
        if (initrd_start && !initrd_below_start_ok &&
                    initrd_start < min_low_pfn << PAGE_SHIFT) {
                printk(KERN_CRIT "initrd overwritten (0x%08lx < 0x%08lx) - "
                    "disabling it.\n",initrd_start,min_low_pfn << PAGE_SHIFT);
                initrd_start = 0;
        }
#endif
        vfs_caches_init_early();
        mem_init();
        kmem_cache_init();
        setup_per_cpu_pageset();
        numa_policy_init();
        if (late_time_init)
               late_time_init();
        calibrate_delay();
        pidmap_init();
        pgtable_cache_init();
        prio_tree_init();
        anon_vma_init();
#ifdef CONFIG_X86
        if (efi_enabled)
               efi_enter_virtual_mode();
#endif
        fork_init(num_physpages);
        proc_caches_init();
        buffer_init();
        unnamed_dev_init();
        key_init();
        security_init();
```

```
        vfs_caches_init(num_physpages);
        radix_tree_init();
        signals_init();
        /* rootfs 的安装可能需要回写页 */
        page_writeback_init();
#ifdef CONFIG_PROC_FS
        proc_root_init();
#endif
        cpuset_init();
        check_bugs();
        acpi_early_init(); /* 在 LAPIC 和 SMP 初始化之前 */
        /* 作其他非__init 的部分，现在已经运行了 */
        rest_init();
}
```

start_kernel()函数负责初始化内核各子系统，最后调用 rest_init()，启动一个叫作 init 的内核线程，继续初始化。

```
/* 我们需要在非__init 函数或者其他函数中完成。
 * root 线程和 init 线程之间存在条件竞争。
 * 在 root 线程执行到 cpu_idle 之前，start_kernel 可能被 free_initmem 扼杀。
 * gcc-3.4 偶然会内联这个函数，所以使用 noinline 的函数类型
 */
static void noinline rest_init(void)
        __releases(kernel_lock)
{
        kernel_thread(init, NULL, CLONE_FS | CLONE_SIGHAND);    /* 启动 init 内核线程 */
        numa_default_policy();
        unlock_kernel();
        preempt_enable_no_resched();
        /* 启动 idle 线程必须执行 schedule()，至少动一下 */
        schedule();
        cpu_idle();
}
```

在 init 内核线程中，将执行下列 init()函数的程序。init 函数负责完成挂接根文件系统、初始化设备驱动和启动用户空间的 init 进程等重要工作。

```
static int init(void * unused)
{
        lock_kernel();
```

```
/* init 可以运行在任何 CPU 上 */
set_cpus_allowed(current, CPU_MASK_ALL);
/* 告诉世界我们就要成为残酷的死神，去扼杀天真的孤儿。
 * 我们希望人们不要对任务所在的队列做错误的假设。
 */
child_reaper = current;

/* Sets up cpus_possible() */
smp_prepare_cpus(max_cpus);

do_pre_smp_initcalls();

fixup_cpu_present_map();
smp_init();
sched_init_smp();

cpuset_init_smp();

/* 在调用 initcalls 之前安装文件系统，因为有些驱动会访问初始化文件。 */
populate_rootfs();

do_basic_setup();    /* 初始化设备驱动 */

/* 检查有没有设置 init。如果有，让它做所有的工作 */
if (!ramdisk_execute_command)
        ramdisk_execute_command = "/init";
/* ramdisk 中没有 init 的话，挂接根文件系统 */
if (sys_access((const char __user *) ramdisk_execute_command, 0) != 0) {
       ramdisk_execute_command = NULL;
       prepare_namespace();
}

/* 现在我们已经完成了初始化启动过程，并且彻底地启动运行起来了。
 * 去掉 initmem 段并且启动用户模式的部分。
 */
free_initmem();
unlock_kernel();
system_state = SYSTEM_RUNNING;
numa_default_policy();
```

```
        if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
                printk(KERN_WARNING "Warning: unable to open an initial console.\n");

         (void) sys_dup(0);
         (void) sys_dup(0);

        if (ramdisk_execute_command) {
                run_init_process(ramdisk_execute_command);
                printk(KERN_WARNING "Failed to execute %s\n",
                            ramdisk_execute_command);
        }

        /* 顺序试探执行，直到有一个成功 */
         * 可以直接执行 shell 替代 init，可以用于恢复系统
         */
        if (execute_command) {
                run_init_process(execute_command);
                printk(KERN_WARNING "Failed to execute %s.  Attempting "
                                "defaults...\n", execute_command);
        }
        run_init_process("/sbin/init");
        run_init_process("/etc/init");
        run_init_process("/bin/init");
        run_init_process("/bin/sh");

        panic("No init found.  Try passing init= option to kernel.");
}
```

对于 Linux 系统来说，挂接根文件系统、初始化设备驱动和启动用户空间的程序是必要的 3 项工作。分析这些重要工作的实现函数，将有助于充分理解内核启动过程。

8.2.5 挂接根文件系统

Linux 能够在内存中虚拟磁盘文件系统，叫作 ramdisk。如果为内核配置了 ramdisk 设备和文件系统，就安装好 ramdisk 文件系统。

```
/* init/initramfs.c */
void __init populate_rootfs(void)
{
     char *err = unpack_to_rootfs(__initramfs_start,
```

```
                    __initramfs_end - __initramfs_start, 0);
        if (err)
                panic(err);
#ifdef CONFIG_BLK_DEV_INITRD
        if (initrd_start) {
                int fd;
                printk(KERN_INFO "checking if image is initramfs...");
                err = unpack_to_rootfs((char *)initrd_start,
                        initrd_end - initrd_start, 1);
                if (!err) {
                        printk(" it is\n");
                        unpack_to_rootfs((char *)initrd_start,
                                initrd_end - initrd_start, 0);
                        free_initrd();
                        return;
                }
                printk("it isn't (%s); looks like an initrd\n", err);
                fd = sys_open("/initrd.image", O_WRONLY|O_CREAT, 700);
                if (fd >= 0) {
                        sys_write(fd, (char *)initrd_start,
                                        initrd_end - initrd_start);
                        sys_close(fd);
                        free_initrd();
                }
        }
#endif
}
```

然后再挂接根文件系统，不过要在初始化设备驱动程序之后执行 prepare_namespace()函数了。它负责为 Linux 系统挂接一个根文件系统。

```
/* init/do_mounts.c */
/* 确定挂接什么文件系统，从哪里挂接，挂接 ramdisk NFS 等文件系统 */
void __init prepare_namespace(void)
{
        int is_floppy;
        mount_devfs();
        if (root_delay) {
                printk(KERN_INFO "Waiting %dsec before mounting root device...\n",
                        root_delay);
```

```
                ssleep(root_delay);
        }
        md_run_setup();
        if (saved_root_name[0]) {
                root_device_name = saved_root_name;
                ROOT_DEV = name_to_dev_t(root_device_name);
                if (strncmp(root_device_name, "/dev/", 5) == 0)
                        root_device_name += 5;
        }
        is_floppy = MAJOR(ROOT_DEV) == FLOPPY_MAJOR;
        if (initrd_load())
                 goto out;
        if (is_floppy && rd_doload && rd_load_disk(0))
                ROOT_DEV = Root_RAM0;
        mount_root();       /* 挂接根文件系统的函数 */
out:        /* 切换根文件系统 */
        umount_devfs("/dev");
        sys_mount(".", "/", NULL, MS_MOVE, NULL);
        sys_chroot(".");
        security_sb_post_mountroot();
        mount_devfs_fs ();
}
```

8.2.6 初始化设备驱动

这里内核子系统已经基本上初始化好了，CPU 子系统已经正常工作，内存管理和进程管理已经正常运转，但是还没有使用任何设备。接下来继续初始化内核设备驱动程序，然后才能访问设备，做系统真正想要做的任务。

```
/* init/main.c */
static void __init do_basic_setup(void)
{
        /* 驱动程序会发送热拔插事件 */
        init_workqueues();
        usermodehelper_init();
        driver_init();

#ifdef CONFIG_SYSCTL
       sysctl_init();
#endif
```

```
        /*网络初始化需要在进程上下文中 */
        sock_init();
        do_initcalls();       /* 执行所有的设备初始化函数 */
}
```

Linux 内核映像把设备驱动程序的初始化函数指针链接成数组，即__initcall_start 和__initcall_end 之间的数据。do_initcalls()函数就是通过调用数组中的函数指针，完成驱动程序的初始化。

```
extern initcall_t __initcall_start[], __initcall_end[]; /* 声明使用外部数组 */
static void __init do_initcalls(void)
{
        initcall_t *call;
        int count = preempt_count();
/* 使用循环语句执行从__initcall_start 到__initcall_end 的所有函数 */
        for (call = __initcall_start; call < __initcall_end; call++) {
                char *msg;
                if (initcall_debug) {
                        printk(KERN_DEBUG "Calling initcall 0x%p", *call);
                        print_fn_descriptor_symbol(": %s()", (unsigned long) *call);
                        printk("\n");
                }
                (*call)();          /* 通过函数指针调用初始化函数 */
                msg = NULL;
                if (preempt_count() != count) {
                        msg = "preemption imbalance";
                        preempt_count() = count;
                }
                if (irqs_disabled()) {
                        msg = "disabled interrupts";
                        local_irq_enable();
                }
                if (msg) {
                        printk(KERN_WARNING "error in initcall at 0x%p: "
                               "returned with %s\n", *call, msg);
                }
        }
        /* 确定 initcall 过程没有导致系统挂起的东西 */
        flush_scheduled_work();
}
```

8.2.7　启动用户空间 init 进程

Linux 系统在挂接根文件系统之后，要执行文件系统的中的应用程序。有些应用程序可能要完成嵌入式系统的。init 进程是通过执行根文件系统中 init 程序启动的。

内核挂接跟文件系统成功以后，将通过 run_init_process()函数执行应用程序。这是一个尝试的过程，如果 execute_command 存在，则执行 execute_command；如果不存在，则顺序执行/sbin/init、/etc/init、/bin/init、/bin/sh，直到有一个执行成功为止。如果都不存在，panic()函数已经等在后面了。

```
/* init/main.c */
static void run_init_process(char *init_filename)
{
        argv_init[0] = init_filename;
        execve(init_filename, argv_init, envp_init);
}
```

run_init_process()函数通过 execve 函数执行应用程序，execue()又调用 do_execuve()。

```
/* arch/arm/kernel/sys_arm.c */
long execve(const char *filename, char **argv, char **envp)
{
        struct pt_regs regs;
        int ret;
        memset(&regs, 0, sizeof(struct pt_regs));
        ret = do_execve((char *)filename, (char __user * __user *)argv,
                   (char __user * __user *)envp, &regs);
        if (ret < 0)
               goto out;
        /* 在寄存器中为用户空间保存 argc */
        regs.ARM_r0 = ret;
        /* 进程启动成功，不再返回到调用函数，而是通过操作内核栈返回用户空间 */
        asm( "add    r0, %0, %1\n\t"
             "mov    r1, %2\n\t"
             "mov    r2, %3\n\t"
             "bl memmove\n\t"    /* copy regs to top of stack */
             "mov    r8, #0\n\t" /* not a syscall */
             "mov    r9, %0\n\t" /* thread structure */
             "mov    sp, r0\n\t" /* reposition stack pointer */
             "b  ret_to_user"
             :
             : "r" (current_thread_info()),
               "Ir" (THREAD_START_SP - sizeof(regs)),
```

```
                "r" (&regs),
                "Ir" (sizeof(regs))
              : "r0", "r1", "r2", "r3", "ip", "lr", "memory");
 out:
              return ret;
}
EXPORT_SYMBOL(execve);
```

execve 函数调用 do_execve()执行用户空间程序。如果执行失败，则返回；如果执行成功，就不再返回。

“黑色经典”系列之《嵌入式 Linux 系统开发技术详解——基于 ARM》

第 9 章　内核调试技术

本章目标

本章介绍了各种 Linux 内核调试方法。内核的调试需要从内核源码本身、调试工具等方面做好准备。通过本章的学习，可以了解不同调试方式的特点和使用方法，根据需要选择不同的内核调试方式。

- 内核调试方法
- 内核打印函数
- 获取内核信息
- 处理出错信息
- 内核源码调试

9.1 内核调试方法

对于庞大的 Linux 内核软件工程，单靠阅读代码查找问题已经非常困难，需要借助调试技术解决 BUG。通过合适的调试手段，可以有效地查找和判断 BUG 的位置和原因。

9.1.1 内核调试概述

当内核运行出现错误的时候，首先要明确定义和可靠地重现这个错误现象。如果一个 BUG 不能重现，修正起来只有凭想象和读代码。内核、用户空间和硬件之间的交互非常，在特定配置、特定机器、特殊负载条件下，运行某些程序可能会产生一个 BUG，其他条件下就不一定产生。这在嵌入式 Linux 系统上很常见，例如：在 X86 平台上运行正常的驱动程序，在 ARM 平台上就可能会出现 BUG。在跟踪 BUG 的时候，掌握的信息越多越好。

内核的 BUG 是多种多样的。可能由于不同原因出现，并且表现形式也多种多样。BUG 范围，从完全不正确的代码（例如：没有在适当的地址存储正确的值）到同步的错误（例如：不适当地对一个共享变量加锁）。它们的表现形式也各种各样，从系统崩溃的错误操作到系统性能差等。

通常 BUG 是一系列事件，内核代码的错误使得用户程序出现错误。例如：一个不待引用数的共享结构体可能引起条件竞争。没有合适的统计，一个进程可以释放这个结构体，但是另外一个进程仍然想要用它。再往下，第二个进程可能会使用通过一个无效的指针访问一个不存在的结构体。这就会导致 NULL 指针废弃、读垃圾数据，如果这个数据还没有被覆盖，也可能基本正常。NULL 指针废弃会产生 oops；垃圾数据导致数据错误（接下来可能是错误的行为或者 oops）；应用程序报告 oops 或者错误的行为。内核开发者必须处理这个错误，知道这个数据是在释放以后访问的，这存在一个条件竞争。修正的方法是为这个结构体添加引用计数，并且可能需要加锁保护。

调试内核很难，实际上内核不同于其他软件工程。内核有操作系统独特的问题，例如：时间管理和条件竞争，这可以使多个线程同时在内核中执行。

因此，调试 BUG 需要有效的调试手段。几乎没有一种调试工具或者方法能够解决全部问题。即使在一些集成测试环境中，也要划分不同测试调试功能，例如：跟踪调试、内存泄漏测试、性能测试等。掌握的调试方法越多，调试 BUG 就越方便。Linux 有很多开放源代码的工具，每一个工具的调试功能专一，所以这些工具的实现一般也比较简单。

9.1.2 学会分析内核源程序

正是由于内核的复杂性，无论使用什么调试手段，都需要熟悉内核源码。只有熟悉了内核各部分的代码实现，才能够找到准确的跟踪点；只有熟悉操作系统的内核机制，才能准确地判断系统运行状态。

对于初学者来说，阅读内核源代码将是非常枯燥的工作。最好先掌握一种搜索工具，学会从源码树中搜索关键词。当能够对内核源代码进行情景分析的时候，你就能感到其中的乐趣了。

调试是无法逃避的任务。进行调试有很多种方法，比如将消息打印到屏幕上、使用调试器，或只是考虑程序执行的情况并仔细地分析问题所在。

在修正问题之前，必须先找出问题的源头。举例来说，对于段错误，需要了解段错误发生在代码的哪一行。一旦发现了代码中出错的行，请确定该方法中变量的值、方法被调用的方式以及关于错误如何发生的详细情况。使用调试器将使找出所有这些信息变得很简单。如果没有调试器可用，还可以使用其他的工具。（请注意：有些 Linux 软件产品中可能并不提供调试器）。

9.1.3　调试方法介绍

内核调试方法很多，主要有以下 4 类。

- 通过打印函数
- 获取内核信息
- 处理出错信息
- 内核源码调试

在调试内核之前，通常需要配置内核的调试选项。图 9.1 给出了“Kernel hacking”配菜单下的各种调试选项。不同的调试方法，需要配置对应的选项。

每一种调试选项针对不同的调试功能。并且不是所有的调试选项在所有的平台上都能够支持。这里介绍一些“Kernel hacking”的调试选项，具体配置使用可以根据情况选择。

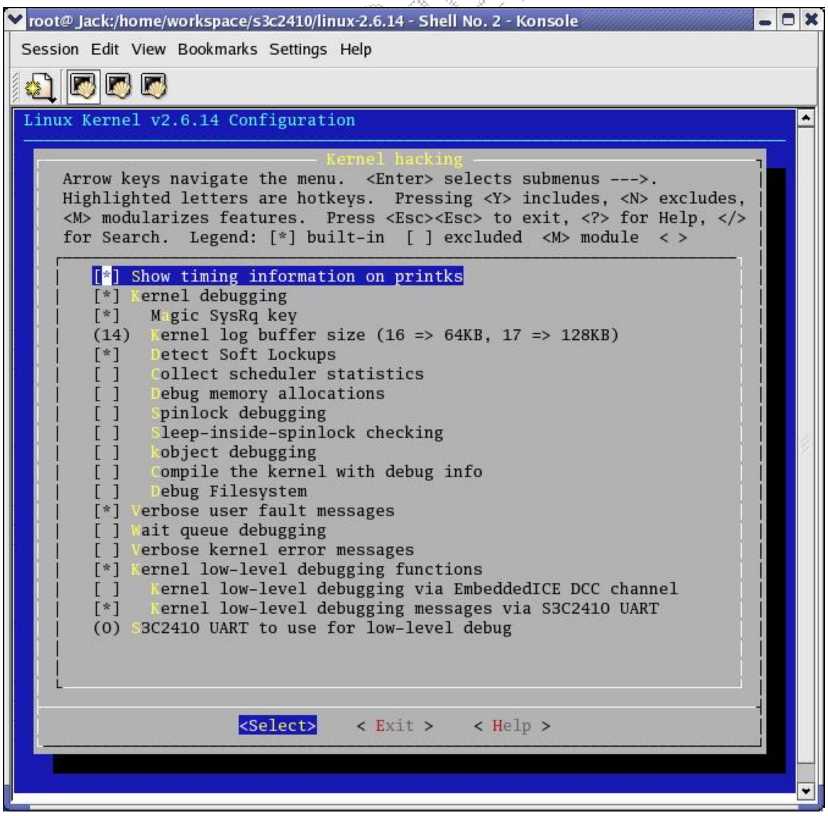


图 9.1 内核调试选项

（1）编译选项“omit-frame-pointer”

CONFIG_FRAME_POINTER 在“Kernel hacking”中缺省的定义为“Y”。在 Makefile 根据这个选项来选择 GCC 编译选项。看看顶层 Makefile 中的这段代码就明白了。

```
    ifdef CONFIG_FRAME_POINTER
    CFLAGS          += -fno-omit-frame-pointer $(call cc-option,-fno-optimize-sibling-calls,)
    else
    CFLAGS          += -fomit-frame-pointer
    endif
```

（2）“Show timing information on printks”

CONFIG_PRINTK_TIME 在 printk 打印的信息中包含时间信息，可以用来测量内核操作之间的时间间隔。

（3）“Kernel debugging”

CONFIG_DEBUG_KERNEL 选择调试内核选项以后，才可以显示有关的内核调试子项。大部分内核调试选项都依赖于它。

（4）“Magic SysRq key”

CONFIG_MAGIC_SYSRQ 使能系统请求键，可以用于系统调试。

（5）“Kernel log buffer size (16 => 64KB, 17 => 128KB)”

CONFIG_LOG_BUF_SHIFT 设置内核日志缓冲区（log_buf）大小，16 表示 64KB，17 表示 128KB。

（6）“Detect Soft Lockups”

CONFIG_DETECT_SOFTLOCKUP 探测内核软死锁状态，例如：有些 BUG 导致内核一直循环超过 10s，而无法调度其他任务执行。一旦探测到死锁状态，内核将打印出当前的堆栈回溯信息。

（7）“Collect scheduler statistics”

CONFIG_SCHEDSTATS 可以在调度器相关的子程序中插入一些代码，采集统计调度器的动作。通过/proc/schedstat 接口可以读取这些信息。

（8）“Debug memory allocations”

CONFIG_DEBUG_SLAB 让内核对内存分配进行有限的验证，这会控制管理空闲的内存，会使 kmalloc()等函数速度变慢。

（9）“Debug preemptible kernel”

CONFIG_DEBUG_PREEMPT 使能内核抢占调试功能。如果在非抢占安全的状况下使用，将打印警告信息。另外，还可以探测抢占技术下溢。

（10）“Spinlock debugging”

CONFIG_DEBUG_SPINLOCK 使能自旋锁（spinlock）调试功能。捕捉自旋锁初始化等方面的错误，结合 NMI watchdog 可以调试死锁。

（11）“Sleep-inside-spinlock checking”

CONFIG_DEBUG_SPINLOCK_SLEEP 检查程序中自旋锁内休眠的情况。因为持有自旋锁休眠将可能引起其他任务等待锁。

（12）“kobject debugging”

CONFIG_DEBUG_KOBJECT 可以把更多 kobject 的调试信息发送到系统日志中。

（13）“Highmem debugging”

CONFIG_DEBUG_HIGHMEM 对高端内存系统进行额外的检查。

（14）“Verbose BUG() reporting (adds 70K)”

CONFIG_DEBUG_BUGVERBOSE 使 BUG()的 panic 报告执行的文件名和调用 BUG()的行号。

（15）“Compile the kernel with debug info”

CONFIG_DEBUG_INFO 使内核映像包含调试信息，可以方便内核源码调试。

（16）“Enable ioremap() debugging”

CONFIG_DEBUG_IOREMAP 这个选项会使内核区分 ioremap 映射的内存和物理内存，并且打印一些回溯信息。

（17）“Debug Filesystem”

CONFIG_DEBUG_FS 支持伪文件系统 debugfs，用来存放调试文件。

（18）“Compile the kernel with frame pointers”

CONFIG_FRAME_POINTER 配置内核编译选项：no-omit-frame-pointer。

（19）“Verbose user fault messages”

CONFIG_DEBUG_USER 在应用程序因为例外崩溃的时候，打印错误信息。

（20）“Wait queue debugging”

CONFIG_DEBUG_WAITQ 使能调试等待队列的功能。

（21）“Verbose kernel error messages”

CONFIG_DEBUG_ERRORS 当内核探测到内部错误的时候，打印调试信息。

（22）“Kernel low-level debugging functions”

CONFIG_DEBUG_LL 包含 printascii、printch、printhex 等调试子程序，方便控制台启动以前的调试工作。

（23）“Kernel low-level debugging via EmbeddedICE DCC channel”

CONFIG_DEBUG_ICEDCC 把调试信息重定向到 EmbeddedICE 的 DCC 通道。这仅对于 ARM9 类型的 ICE 仿真器可以使用。

（24）“Kernel low-level debugging messages via footbridge serial port”

CONFIG_DEBUG_DC21285_PORT 把调试信息重定向到 DC21285（Footbridge）串口。这仅对于特定的硬件平台可用。

（25）“Kernel low-level debugging messages via S3C2410 UART”

CONFIG_DEBUG_S3C2410_PORT 把调试信息重定向到 S3C2410 串口。这仅对于 S3C2410 平台可用。

9.2 内核打印函数

嵌入式系统一般都可以通过串口与用户交互。大多数 Bootloader 可以向串口打印信息，并且接收命令。内核同样可以向串口打印信息。但是在内核启动过程中，不同阶段的打印函数不同。分析这些打印函数的实现，可以更好地调试内核。

9.2.1 内核映像解压前的串口输出函数

从启动信息可以看出，zImage 在解压之前就向串口输出提示信息了。回顾一下第 7 章介绍的 Linux 启动过程，decompresss_kernel()函数调用了 putstr()函数，直接向串口打印内核解压的信息。

putstr()函数实现了向串口输出字符串的功能。因为不同的处理器可以有不同的串口控制器，所以 putstr()函数的实现依赖于硬件平台。分析一下 S3C2410 平台中 putstr()函数的实现，它是在头文件 uncompress.h 中实现的。

```
/* include/asm/arch/uncompress.h */
/* 写 UART 寄存器的函数实现 */
static __inline__ void
uart_wr(unsigned int reg, unsigned int val)
{
     volatile unsigned int *ptr;
     ptr = (volatile unsigned int *)(reg + uart_base);
     *ptr = val;
}
/* 读 UART 寄存器的函数实现 */
static __inline__ unsigned int
uart_rd(unsigned int reg)
{
     volatile unsigned int *ptr;
     ptr = (volatile unsigned int *)(reg + uart_base);
     return *ptr;
}
/* 输出单个字符的函数 */
static void
putc(char ch)
{    /* 要处理 UART 运行在 FIFO 模式的情况，不能因为等待 TX 信号而停止执行。 */
     int cpuid = S3C2410_GSTATUS1_2410;
     if (ch == '\n')
            putc('\r');    /* 扩展成回车换行符\r\n */
     if (uart_rd(S3C2410_UFCON) & S3C2410_UFCON_FIFOMODE) { /* FIFO 模式 */
```

```
            int level;
            while (1) { /* 如果FIFO模式，等待FIFO非满的状态 */
                  level = uart_rd(S3C2410_UFSTAT);
                  if (cpuid == S3C2410_GSTATUS1_2440) {
                        level &= S3C2440_UFSTAT_TXMASK;
                        level >>= S3C2440_UFSTAT_TXSHIFT;
                  } else {
                        level &= S3C2410_UFSTAT_TXMASK;
                        level >>= S3C2410_UFSTAT_TXSHIFT;
                  }
                  if (level < FIFO_MAX)
                        break;
            }
      } else {   /* 如果不使用FIFO的模式，就等待TX状态 */
            while ((uart_rd(S3C2410_UTRSTAT) & S3C2410_UTRSTAT_TXE) \
                   != S3C2410_UTRSTAT_TXE);
      }
      /* 向发送寄存器写字节操作 */
      uart_wr(S3C2410_UTXH, ch);
}
/* 输出字符串的函数 */
static void
putstr(const char *ptr)
{
      for (; *ptr != '\0'; ptr++) {
            putc(*ptr);          /* 调用输出单个字符的函数 */
      }
}
```

这些函数的实现比较简单，基本上就是 RAW 设备操作的方式。直接对串口寄存器读写，实现串口输入输出的功能。

9.2.2 内核错误报告子程序

在内核解压完成，跳转到 vmlinux 映像入口。这时还没有初始化控制台设备，但是执行系统初始化的过程中也可能出现严重的错误，导致系统崩溃。怎样才能报告这种错误信息呢？可以通过 printascii 子程序来向串口打印。

printascii、printhex8 等子程序包含在 arch/arm/kernel/debug.S 文件中。如果要编译链接这些子程序，需要内核使能图 9.1 中的“Kernel low-level debugging functions”选项。

printascii 子程序实现向串口打印字符串的功能，printhex 也调用了 printascii 子程序来显

示数字。在 printascii 子程序中，调用了宏（macro）：addruart、waituart、senduart、busyuart，这些宏都是在 include/asm/arch/debug-macro.S 中定义的。这里简单分析一下 printascii 的程序。

```
/* arch/arm/kernel/debug.S */
            .ltorg
ENTRY(printascii)
            addruart r3
            b   2f
1:          waituart r2, r3
            senduart r1, r3
            busyuart r2, r3
            teq r1, #'\n'
            moveq   r1, #'\r'
            beq 1b
2:          teq r0, #0
            ldrneb  r1, [r0], #1
            teqne   r1, #0
            bne 1b
            mov pc, lr
```

其中一个应用就是__error_a 报错子程序，报告系统平台类型（machine type）的检查结果。这里就是调用 printascii 打印信息的。

```
/* arch/arm/kernel/head.S */
     .type  __error_a, %function
__error_a:
#ifdef CONFIG_DEBUG_LL
     mov    r4, r1              @ preserve machine ID
     adr    r0, str_a1
     bl     printascii
     mov    r0, r4
     bl     printhex8
     adr    r0, str_a2
     bl printascii
     adr    r3, 3f
     ldmia  r3, {r4, r5, r6}        @ get machine desc list
     sub    r4, r3, r4          @ get offset between virt&phys
     add    r5, r5, r4          @ convert virt addresses to
     add    r6, r6, r4          @ physical address space
1:   ldr    r0, [r5, #MACHINFO_TYPE]    @ get machine type
     bl     printhex8
```

```
        mov     r0, #'\t'
        bl      printch
        ldr     r0, [r5, #MACHINFO_NAME]    @ get machine name
        add     r0, r0, r4
        bl      printascii
        mov     r0, #'\n'
        bl      printch
        add     r5, r5, #SIZEOF_MACHINE_DESC    @ next machine_desc
        cmp     r5, r6
        blo     1b
        adr     r0, str_a3
        bl      printascii
        b       __error
str_a1:     .asciz  "\nError: unrecognized/unsupported machine ID (r1 = 0x"
str_a2:     .asciz  ").\n\nAvailable machine support:\n\nID (hex)\tNAME\n"
str_a3:     .asciz  "\nPlease check your kernel config and/or bootloader.\n"
        .align
#endif
```

Linux 内核通过结构体来描述硬件平台，这些结构体在链接的时候组成一个列表。内核启动过程时，首先要从这些列表中查找对应的系统平台类型。这项工作由lookup_machine_type 子程序完成。从 u-boot 的 bootm 命令代码分析可以看出，Bootloader 可以通过寄存器把系统平台类型（Architecture Number）传递过来。如果找不到相应的系统号，就调用__error_a 报错："Error: unrecognized/unsupported machine ID (r1 = 0x)"。

```
/* arch/arm/kernel/head.S */
/* 查找平台系统号，这时还没有启动 MMU，不能使用__arch_info 列表的地址
* r1 = 平台系统号
* r5 = 找到的 mach_info 在物理内存中的指针
*/
        .type   __lookup_machine_type, %function
__lookup_machine_type:
        adr     r3, 3b
        ldmia   r3, {r4, r5, r6}
        sub     r3, r3, r4          @ get offset between virt&phys
        add     r5, r5, r3          @ convert virt addresses to
        add     r6, r6, r3          @ physical address space
1:      ldr     r3, [r5, #MACHINFO_TYPE]    @ get machine type
        teq     r3, r1              @ matches loader number?
        beq     2f              @ found
        add     r5, r5, #SIZEOF_MACHINE_DESC    @ next machine_desc
```

```
        cmp     r5, r6
        blo     1b
        mov     r5, #0              @ unknown machine
2:      mov     pc, lr

/* 提供上面函数的一个 C-API 版本 */
ENTRY(lookup_machine_type)
        stmfd   sp!, {r4 - r6, lr}
        mov     r1, r0
        bl      __lookup_machine_type
        mov     r0, r5
        ldmfd   sp!, {r4 - r6, pc}
```

start_kernel()调用了 setup_arch()函数，setup_arch()函数中又顺序调用了 setup_processor()和 setup_machine()函数。这 2 个函数中分别检查处理器类型和系统平台类型，通常系统平台类型不正确是最常见的错误。lookup_machine_type 子程序就是在 setup_machine()函数中调用的。

根据上面的分析，我们可以通过 printascii 打印出来的信息判断出错原因，也可以添加其他的打印信息，调试系统启动过程。

9.2.3 内核打印函数

Linux 内核标准的系统打印函数是 printk。printk 函数具有极好的健壮性，不受内核运行条件的限制，在系统运行期间都可以使用。printk 日志级别如表 9.1 所示。

这些级别有助于内核控制信息的紧急程度，判断是否向串口输出等。正如 printk 函数的日志级别，printk 的函数的实现也比较复杂。printk 函数不是直接向控制台设备或者串口直接打印信息，而是把打印信息先写到缓冲区里面。分析一下 printk 函数的代码实现。

表 9.1 printk 日志级别

日 志 级 别	说　　明	日 志 级 别	说　　明
KERN_EMERG	紧急情况，系统可能会死掉	KERN_WARNING	警告信息
KERN_ALERT	需要立即响应的问题	KERN_NOTICE	普通但是可能有用的信息
KERN_CRIT	重要情况	KERN_INFO	情报信息
KERN_ERR	错误信息	KERN_DEBUG	调试信息

```
/* kernel/printk.c */
/* 不指定级别的 printk 函数用这个缺省级别.. */
#define DEFAULT_MESSAGE_LOGLEVEL 4 /* KERN_WARNING 级别 */
......
#define MINIMUM_CONSOLE_LOGLEVEL 1 /* 控制台可以使用的最小级别数 */
#define DEFAULT_CONSOLE_LOGLEVEL 7 /* 任何比 KERN_DEBUG 更严重级别的信息都显示 */
```

```
DECLARE_WAIT_QUEUE_HEAD(log_wait);
int console_printk[4] = {  /* 定义控制台的缺省打印级别 */
      DEFAULT_CONSOLE_LOGLEVEL,/* console_loglevel */
      DEFAULT_MESSAGE_LOGLEVEL,/* default_message_loglevel */
      MINIMUM_CONSOLE_LOGLEVEL,/* minimum_console_loglevel */
      DEFAULT_CONSOLE_LOGLEVEL,/* default_console_loglevel */
};
……
/* 这是 printk 函数的实现，它可以在任何上下文中调用，不会出问题。
 * 对控制台操作之前，先测试 console_sem 信号灯。
 * 如果成功，把输出信息记录日志并且调用控制台驱动;
 * 如果失败，把输出信息写到日志缓冲区中，立即返回。
 */
asmlinkage int printk(const char *fmt, ...)
{
      va_list args;
      int r;
      va_start(args, fmt);     /* 使用变参 */
      r = vprintk(fmt, args);  /* vprintk 函数完成打印任务 */
      va_end(args);
      return r;
}
/* 处理器现在持有 logbuf_lock 这个锁 */
static volatile unsigned int printk_cpu = UINT_MAX;
/* vprintk 函数的实现 */
asmlinkage int vprintk(const char *fmt, va_list args)
{
      unsigned long flags;
      int printed_len;
      char *p;
      static char printk_buf[1024];    /* printk_buf 是临时缓冲区 */
      static int log_level_unknown = 1;

      preempt_disable();
      if (unlikely(oops_in_progress) && printk_cpu == smp_processor_id())
             /* 如果处理器在调用 printk()的过程中崩溃，确保不会死锁 */
             zap_locks();
      /* This stops the holder of console_sem just where we want him */
      spin_lock_irqsave(&logbuf_lock, flags);
```

```
        printk_cpu = smp_processor_id();
        /* 把输出信息写到临时缓冲区 */
        printed_len = vscnprintf(printk_buf, sizeof(printk_buf), fmt, args);
        /* 把输出信息复制到 log_buf。需要插入对应的日志级别标记 */
        for (p = printk_buf; *p; p++) {
                if (log_level_unknown) {  /* log_level_unknown 发出换行的信号 */
                        if (printk_time) {
                                int loglev_char;
                                char tbuf[50], *tp;
                                unsigned tlen;
                                unsigned long long t;
                                unsigned long nanosec_rem;
                                /* 在输出之前加上日志级别标志，例如: <1> */
                                if (p[0] == '<' && p[1] >='0' &&
                                    p[1] <= '7' && p[2] == '>') {
                                      loglev_char = p[1];
                                      p += 3;
                                      printed_len += 3;
                                } else {
                                      loglev_char = default_message_loglevel + '0';
                                }
                                t = printk_clock();
                                nanosec_rem = do_div(t, 1000000000);
                                tlen = sprintf(tbuf,
                                            "<%c>[%5lu.%06lu] ",
                                            loglev_char,
                                            (unsigned long)t,
                                            nanosec_rem/1000);
                                for (tp = tbuf; tp < tbuf + tlen; tp++)
                                            emit_log_char(*tp);
                                printed_len += tlen - 3;
                        } else {
                                if (p[0] != '<' || p[1] < '0' ||
                                     p[1] > '7' || p[2] != '>') {
                                      emit_log_char('<');
                                      emit_log_char(default_message_loglevel + '0');
                                      emit_log_char('>');
                                }
                                printed_len += 3;
```

```
                    }
                    log_level_unknown = 0;
                    if (!*p)
                          break;
              }
              emit_log_char(*p);
              if (*p == '\n')
                     log_level_unknown = 1;
       }
       if (!cpu_online(smp_processor_id())) {
             /* 有时候当前 CPU 还没有使能。直到 CPU 打开的时候，才调用控制台驱动程序。 */
             printk_cpu = UINT_MAX;
             spin_unlock_irqrestore(&logbuf_lock, flags);
             goto out;
       }
       if (!down_trylock(&console_sem)) {
             console_locked = 1;
             /* 驱动程序可用时,释放 spinlock,通过 release_console_sem()打印信息 */
             printk_cpu = UINT_MAX;
             spin_unlock_irqrestore(&logbuf_lock, flags);
       console_may_schedule = 0;
       release_console_sem();
       } else {  /*驱动程序被占用时，释放 spinlock，让别人打印 */
              printk_cpu = UINT_MAX;
              spin_unlock_irqrestore(&logbuf_lock, flags);
       }
out:
       preempt_enable();
       return printed_len;
}
EXPORT_SYMBOL(printk);  /* 输出函数符号，内核其他程序可以调用 */
EXPORT_SYMBOL(vprintk);
```

printk()函数调用了 vprintk()函数，vprintk()函数有调用了 emit_log_char()函数处理数据。emit_log_char()又把数据怎么处理了呢？

```
static void emit_log_char(char c)
{
       LOG_BUF(log_end) = c; /* 写到环形日志数据末尾 */
       log_end++;
```

```
        if (log_end - log_start > log_buf_len)
                log_start = log_end - log_buf_len;
        if (log_end - con_start > log_buf_len)
                con_start = log_end - log_buf_len;
        if (logged_chars < log_buf_len)
                logged_chars++;
}
```

emit_log_char()函数把数据写到环形日志缓冲区 log_buf []中了。然后调用控制台驱动程序，在控制台显示这些信息。

```
/* 调用控制台驱动程序输出 log_buf 信息 */
static void __call_console_drivers(unsigned long start, unsigned long end)
{
        struct console *con;
        for (con = console_drivers; con; con = con->next) {
                if ((con->flags & CON_ENABLED) && con->write)
                       con->write(con, &LOG_BUF(start), end - start); /* 控制台写操作 */
        }
}

/* 从 start 写到 end，仅包括一遍 */
static void _call_console_drivers(unsigned long start,
                              unsigned long end, int msg_log_level)
{
        if (msg_log_level < console_loglevel &&
                     console_drivers && start != end) {
                if ((start & LOG_BUF_MASK) > (end & LOG_BUF_MASK)) {
                     /* 带换行写 */
                     __call_console_drivers(start & LOG_BUF_MASK,
                                     log_buf_len);
                     __call_console_drivers(0, end & LOG_BUF_MASK);
                } else {
                     __call_console_drivers(start, end);
                }
        }
}

/* 调用控制台驱动程序，写出从 log_buf[start]到 log_buf[end-1]数据，必须持有 console_sem */
static void call_console_drivers(unsigned long start, unsigned long end)
```

```
{
        unsigned long cur_index, start_print;
        static int msg_level = -1;
        if (((long)(start - end)) > 0)
                BUG();
        cur_index = start;
        start_print = start;
        while (cur_index != end) {
                if (   msg_level < 0 &&
                        ((end - cur_index) > 2) &&
                        LOG_BUF(cur_index + 0) == '<' &&
                        LOG_BUF(cur_index + 1) >= '0' &&
                        LOG_BUF(cur_index + 1) <= '7' &&
                        LOG_BUF(cur_index + 2) == '>')
                {
                        msg_level = LOG_BUF(cur_index + 1) - '0';
                        cur_index += 3;
                        start_print = cur_index;
                }
                while (cur_index != end) {
                        char c = LOG_BUF(cur_index);
                        cur_index++;
                        if (c == '\n') {
                                if (msg_level < 0) {
                                        /* printk()已经给出了日志级别标志，要按照对应格式 */
                                        msg_level = default_message_loglevel;
                        }
                        _call_console_drivers(start_print, cur_index, msg_level);
                        msg_level = -1;
                        start_print = cur_index;
                        break;
                }
            }
        }
        _call_console_drivers(start_print, end, msg_level);
}
void release_console_sem(void)
{
        unsigned long flags;
```

```
        unsigned long _con_start, _log_end;
        unsigned long wake_klogd = 0;

        for ( ; ; ) {
                spin_lock_irqsave(&logbuf_lock, flags);
                wake_klogd |= log_start - log_end;
               if (con_start == log_end)
                     break;            /* Nothing to print */
               _con_start = con_start;
               _log_end = log_end;
               con_start = log_end;       /* Flush */
               spin_unlock(&logbuf_lock);
               call_console_drivers(_con_start, _log_end);
               local_irq_restore(flags);
        }
        console_locked = 0;
        console_may_schedule = 0;
        up(&console_sem);
        spin_unlock_irqrestore(&logbuf_lock, flags);
        if (wake_klogd && !oops_in_progress && waitqueue_active(&log_wait))
              wake_up_interruptible(&log_wait);
    }
    EXPORT_SYMBOL(release_console_sem);
```

release_console_sem()函数调用 call_console_drivers()，call_console_drivers()函数再调用_call_console_drivers()函数，_call_console_drivers()函数再调用__call_console_drivers()函数，最后__call_console_drivers()调用控制台驱动程序的写操作函数 con->write()。这样 log_buf[]中的数据就全部输出到控制台上了。

release_console_sem()函数可以在任意上下文中调用，所以 log 信息能够及时地打印到控制台上。

9.3 获取内核信息

Linux 内核提供了一些与用户空间通信的机制，大部分驱动程序与用户空间的接口都可以作为获取内核信息的手段。另外内核也有专门的调试机制。

9.3.1 系统请求键

系统请求键可以使 Linux 内核回溯跟踪进程，当然这要在 Linux 的键盘仍然可用的前提下，并且 Linux 内核已经支持 MAGIC_SYSRQ 功能模块。

大多数系统平台（特别是 X86）都已经实现了系统请求键功能，它是在 drivers/char/sysrq.c 中实现的。在配置内核的时候需要选择图 8.1 的“Magic SysRq key”菜单选项，使能配置选项 CONFIG_MAGIC_SYSRQ。

使用这项功能，必须是在文本模式的控制台上，并且启动 CONFIG_MAGIC_SYSRQ。

SysRq（系统请求）键是复合键【Alt+SysRq】，大多数键盘的 SysRq 和 PrtSc 键是复用的。

按住 SysRq 复合键，再输入第三个命令键，可以执行相应的系统调试命令。例如：输入 t 键，可以得到当前运行的进程和所有进程的堆栈跟踪。回溯跟踪将被写到 /var/log/messages 文件中。如果内核都配置好了，系统应该已经转换了内核的符号地址。

但是，在串口控制台上不能使用 SysRq 复合键。可以先发送一个“BREAK”，在 5s 之内输入系统请求命令键。

另外，有些硬件平台也不能使用 SysRq 复合键。不过，各种目标板都可以通过/proc 接口进入系统请求状态。

```
$ echo t > /proc/sysrq-trigger
```

表 9.2 列出系统请求键的命令解释。更多信息可以查阅内核文档 Documentation/sysrq.txt。

表 9.2　　　　系统请求键命令

键　命　令	说　　明
SysRq-b	重起机器
SysRq-e	给 init 之外的所有进程发送 SIGTERM 信号
SysRq-h	在控制台上显示 SysRq 帮助
SysRq-i	给 init 之外的所有进程发送 SIGKILL 信号
SysRq-k	安全访问键：杀掉这个控制台上所有进程
SysRq-l	给包括 init 的所有进程发送 SIGKILL 信号
SysRq-m	在控制台上显示内存信息
SysRq-o	关闭机器
SysRq-p	在控制台上显示寄存器
SysRq-r	关闭键盘的原始模式
SysRq-s	同步所有挂接的磁盘
SysRq-t	在控制台上显示所有的任务信息
SysRq-u	卸载所有已经挂载的磁盘

神奇的系统请求键是辅助调试或者拯救系统的重要方法。它为控制台上的任何用户提供了强大的功能。当出现系统宕机或者运行状态不正常的时候，通过系统请求键可以查询当前进程执行的状态，从而判断出错的进程和函数。

9.3.2　通过/proc 接口

proc 文件系统是一种伪文件系统。实际上，它并不占用存储空间，而是系统运行时在内存中建立的内核状态映射，可以瞬时地提供系统的状态信息。

在用户空间可以作为文件系统挂接到/proc 目录下，提供给用户访问。可以通过 Shell 命令挂接，也可以在/etc/fstab 中做出相应的设置。

```
$ mout –t proc proc /proc
```

通过 proc 文件系统可以查看运行中的内核，查询和控制运行中的进程和系统资源等状态。这对于监控性能、查找系统信息、了解系统是如何配置的以及更改该配置很有用。

在用户空间，可以直接访问/proc 目录下的条目，读取信息或者写入命令。但是不能使用编辑器打开修改/proc 条目，因为在编辑过程中，同步保存的数据将是不完整的命令。

命令行下使用 echo 命令，从命令行将输出重定向至/proc 下指定条目中。例如关闭系统请求键功能的命令：

```
$ echo 0 > /proc/sys/kernel/sysrq
```

命令行下查看/proc 目录下的条目信息，应该使用命令行下的 cat 命令。例如：

```
$ cat /proc/cpuinfo
```

另外，/proc 接口的条目可以作为普通的文件打开访问。这些文件也有访问的权限限制，大部分条目是只读的，少数用于系统控制的条目具有写操作属性。在应用程序中，可以通过 open()、read()、write()等函数操作。

/proc 中的每个条目都有一组分配给它的非常特殊的文件访问权限，并且每个文件属于特定的用户标识。这一点实现得非常仔细，从而提供给管理员和用户正确的功能。这些特定的访问权限如下。

（1）只读权限：任何用户都不能对该文件进行写操作，它用于获取系统信息。

（2）root 写权限：如果/proc 中的某个文件是可写的，则通常只能由 root 用户来写。

（3）root 读权限：有些文件对一般系统用户是不可见的，而只对 root 用户是可见的。

（4）其他权限：可能有不同于以上常见的 3 种访问权限的组合。

就具体/proc 条目的功能而言，每一个条目的读写操作在内核中都有特定的实现。当查看/proc 目录下文件时，会发现有些文件是可读的，可以从中读出内核的特定的信息；有些文件是可写的，可以写入特定的配置和控制命令。

Linux 的一些系统工具就是通过/proc 接口读取信息的。例如：top 命令就是读取/proc 接口下相关条目的信息，实时地显示当前运行中的进程和系统负载。

要获得/proc 文件的所有信息，一个最佳来源就是 Linux 内核源代码本身，它包含了一些非常优秀的文档。

9.3.3 通过/sys 接口

Sysfs 文件系统是 Linux 2.6 内核新增加的文件系统。它也是一种伪文件系统，是在内存中实现的文件系统。它可以把内核空间的数据、属性、链接等东西输出到用户空间。

在 Linux 2.6 内核中，sysfs 和 kobject 是紧密结合的，成为驱动程序模型的组成部分。

当加载或者卸载 kobject 的时候，需要注册或者注销操作。当注册 kobject 时，注册函数除了把 kobject 插入到 kset 链表中，还要在 sysfs 中创建对应的目录。反过来，当注销 kobject 时，注销函数也会删除 sysfs 中相应的目录。

通常 sysfs 文件系统要挂接到/sys 目录下，提供给用户空间访问。可以通过 Shell 命令挂接，也可以在/etc/fstab 中做出相应的设置。

```
$ mount –t sysfs sysfs /sys
```

sysfs 文件系统的目录组织结构反映了内核数据结构的关系。/sys 的目录结构下应该包含以下子目录。

```
block/ bus/ class/ devices/ firmware/ net/
```

devices/目录下的目录树代表设备树，它直接映射了内核内部的设备树（它按照 device 结构体的层次关系）。

bus/目录包含内核各种总线类型的目录。每一种总线目录包含两个子目录：devices/和 drives/。

devices/目录包含了系统探测到的每一个设备的符号链接，指向 sysfs 文件系统的 root/目录下的设备。

drivers/目录包含在特定总线结构上为每一个加载的设备驱动创建的子目录。

class/目录包含设备接口类型的目录，当然在类型子目录下还有设备接口的子目录。

```
class/
'-- input
      |-- devices
      |-- drivers
      |-- mouse
      '-- evdev
     ……
```

为了方便使用 sysfs，下面介绍一些 sysfs 的编程接口。

第一个方面是属性。属性能够以文件系统的正常文件形式输出到用户空间。Sysfs 文件系统间接调用属性定义的函数操作，提供读写内核属性的方法。

属性应该是 ASCII 文本文件，每个文件只能有一个值。可能这样效率不高，可以通过相同类型的数组来表示。

不赞成使用混合类型、多行数据格式和奇异的数据格式。这样做可能使代码得不到认可。

简单的属性定义示例：

```
struct attribute {
            char                    * name;
            mode_t                  mode;
};
int sysfs_create_file(struct kobject * kobj, struct attribute * attr);
void sysfs_remove_file(struct kobject * kobj, struct attribute * attr);
```

定义空洞的属性是没有用的，所以最好针对特定的目标类型添加自己的结构体属性或者封装好的函数。

例如，设备驱动程序可以定义下面的结构体 device_attribute。

```
struct device_attribute {
          struct attribute      attr;
          ssize_t (*show)(struct device * dev, char * buf);
          ssize_t (*store)(struct device * dev, const char * buf);
};
int device_create_file(struct device *, struct device_attribute *);
void device_remove_file(struct device *, struct device_attribute *);
```

使用下面的宏定义，也可以预先定义辅助定义设备属性的宏。

```
#define DEVICE_ATTR(_name, _mode, _show, _store)     \
struct device_attribute dev_attr_##_name = {          \
           .attr = {.name  = __stringify(_name) , .mode   = _mode },     \
           .show   = _show,                          \
           .store  = _store,                       \
};
```

举例说明使用上面的宏来定义属性。

```
static DEVICE_ATTR(foo, S_IWUSR | S_IRUGO, show_foo, store_foo);
```

等价于：

```
static struct device_attribute dev_attr_foo = {
          .attr = {
              .name = "foo",
              .mode = S_IWUSR | S_IRUGO,
      },
      .show = show_foo,
      .store = store_foo,
};
```

第二个方面是子系统操作函数。当子系统定义了一个属性类型时，必须实现一些 sysfs 操作函数。当应用程序调用 read/write 函数时，通过这些子系统函数显示或者保存属性值。

```
struct sysfs_ops {
        ssize_t (*show)(struct kobject *, struct attribute *, char *);
        ssize_t (*store)(struct kobject *, struct attribute *, const char *);
};
```

当读或者写这个 sysfs 文件时，sysfs 调用对应的函数。然后，把通用的 kobject 结构体和结构体属性指针转换成适当的指针类型，并且调用相关的函数。

举例说明

```
#define to_dev_attr(_attr) container_of(_attr, struct device_attribute, attr)
#define to_dev(d) container_of(d, struct device, kobj)

static ssize_t
dev_attr_show(struct kobject * kobj, struct attribute * attr, char * buf)
{
        struct device_attribute * dev_attr = to_dev_attr(attr);
        struct device * dev = to_dev(kobj);
        ssize_t ret = 0;
        if (dev_attr->show)
                ret = dev_attr->show(dev, buf);
        return ret;
}
```

要读写属性，还要声明和实现 show() 和 store()函数。这两个函数的声明如下：

```
ssize_t (*show)(struct device * dev, char * buf);
ssize_t (*store)(struct device * dev, const char * buf);
```

读写函数的操作主要是数据缓冲区的读写操作，一个最简单的设备属性实现的例子。

```
static ssize_t show_name(struct device *dev, struct device_attribute *attr,
char *buf)
{
       return snprintf(buf, PAGE_SIZE, "%s\n", dev->name);
}
static ssize_t store_name(struct device * dev, const char * buf)
{
       sscanf(buf, "%20s", dev->name);
       return strnlen(buf, PAGE_SIZE);
}
static DEVICE_ATTR(name, S_IRUGO, show_name, store_name);
```

9.3.4 通过 ioctl 方法

ioctl 是对一个文件描述符响应的系统调用，它可以实现特殊命令操作。ioctl 可以替代/proc 文件系统，实现一些调试的命令。

使用 ioctl 获取信息比/proc 麻烦一些，因为通过应用程序的 ioctl 函数调用并且显示结果必须编写、编译一个应用程序，并且与正在测试的模块保持一致。反过来，驱动程序代码比实现/proc 文件相对简单一点。

大多数时候 ioctl 是获取信息的最好方法，因为它比读/proc 运行得快。假如数据必须在打印到屏幕上之前处理，以二进制格式获取数据将比读一个文本文件效率更高。另外，ioctl 不需要把数据分割成小于一个页的碎片。

ioctl 还有另外一个优点，就是信息获取命令可以保留在驱动程序中，即使已经完成调试工作。不像/proc 文件，在目录下所有人都可以看到。

在内核空间，ioctl 驱动程序函数原型如下。

```
int (*ioctl) (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
```

这个 inode 和 filp 指针是应用程序传递的文件描述符的值和与传递给 open 函数同样的参数。这个 cmd 参数是从用户空间未加修改传递过来的，可选的参数 arg 以无符号长整数传递，不管是使用整数还是指针。如果调用这个函数的时候不传递第 3 个参数，驱动程序接收的 arg 是未定义的。因为对于额外的参数的类型检查已经关闭，编译器不会警告一个非法的参数传递给 ioctl，并且任何相关的 BUG 都将很难查找。

大多数 ioctl 实现包含了一个大的 switch 语句，可以根据 cmd 参数选择适当的操作。不同的命令有不同的数值，可以通过符号名简化编程。这些符号名通过预编译定义。定制的驱动可以在头文件中声明这些符号。用户程序也必须包含这些头文件，以便使用这些符号。

用户空间可以使用 ioctl 系统调用。

```
int ioctl(int fd, unsigned long cmd, ...);
```

原型函数的省略号标志是说这个函数可以传递数量可变的参数。在实际系统中，系统调用不能用数量可变的参数。系统调用必须使用定义好的原型，因为用户可以通过硬件操作来访问。因此，这些省略号不代表变参，而是一个可选参数，传统上定义为 char *argp。原型的省略号可以防止编译过程的类型检查。第 3 个参数的本质依赖于特定的控制命令（第 2 个参数）。有些命令没有参数，有些取整型参数，有些取数据指针。使用指针可以把任意数据传递给 ioctl 函数；设备就可以与用户空间交互任意大小的数据块了。

ioctl 函数的不规范性使内核开发者并不喜欢。每一个 ioctl 命令是一个分离的非正式的系统调用，并且没有办法按照易于理解的方式整理，也很难使这些不规范的 ioctl 参数在所有的系统上都能工作。例如：用户空间进程运行 32 位模式的 64 位系统。这导致强烈需要实现其他方式的多种控制操作。可行的方式包括数据流中嵌入命令或者使用虚拟文件系统，sysfs 或者驱动程序相关的文件系统。但是，事实上，ioctl 仍然是对设备操作最简单和最直接的选择。

9.4 处理出错信息

当系统出现错误时，内核有两个基本的错误处理机制：oops 和 panic。

9.4.1 oops 信息

尽管有了各种调试方法，系统或者驱动程序的一些 BUG 仍可能直接导致系统出错，打印出 oops 信息。通常 oops 发生以后，系统处于不稳定状态，可能崩溃，也可能继续运行。

（1）oops 消息包含系统错误的详细信息

通常 oops 信息中包含当前进程的栈回溯和 CPU 寄存器的内容。分析在发生崩溃时发送到系统控制台的 oops 消息，这是 Linux 调试系统崩溃的传统方法。oops 信息是机器指令级的，可以说是很难懂。ksymoops 工具可以将机器指令转换为代码并将堆栈值映射到内核符号。在很多情况下，这些信息就足够确定错误的可能原因。

分析 oops 信息是一项很艰苦的工作，先来看看这些信息吧。

```
Oops: machine check, sig: 7
NIP: C000F290 XER: 20000000 LR: C000F0F0 SP: C013F940 REGS: c013f890 TRAP: 0200
MSR: 00009030 EE: 1 PR: 0 FP: 0 ME: 1 IR/DR: 11
TASK = c013e020[0] 'swapper' Last syscall: 120
last math 00000000 last altivec 00000000
GPR00: 00000000 C013F940 C013E020 000001F5 C500F200 C3A89000 00000002 C023BFA8
GPR08: 00000007 00000570 0000017B 0000015C 84002022 1002B4DC 00000000 00000000
GPR16: 00000000 00000000 00000000 00000000 00001032 0013FA90 00000000 C00047CC
GPR24: C0150000 000003C0 C07368C0 C013F9C8 000005EE C3A89000 C0160000 C0160000
Call backtrace:
C00334C8 C0160000 C000EE4C C00ACE60 C00A9584 C00AD258 C00AD008
C00A879C C00057A4 C0005860 C00047CC 00000020 C00C1404 C00C146C
C00A8C08 C00CE3C8 C00C59A4 C00DA4A4 C00D9068 C00DA608 C00D9340
C00E9224 C00E7A54 C00EFDF4 C00F032C C00D62CC C00D6504 C00C6060
C00C6214 C00C6384 C001B820 C00058C8 C00047CC
Kernel panic: Aiee, killing interrupt handler!
Warning (Oops_read): Code line not seen, dumping what data is available
```

其中打印出了处理器寄存器的值，还有进程（Task）和栈回溯（Call Trace）信息。对照 System.map，完全可以分析一下的。不过，还有一个更好的工具来辅助分析。

（2）使用 ksymoops 转换 oops 信息

Ksymoops 工具可以翻译 oops 信息，从而分析发生错误的指令，并显示一个跟踪部分表明代码如何被调用。它是根据内核映像的 System.map 来转换的，因此，必须提供正在运行的内核映像的 System.map 文件。

关于如何使用 ksymoops，内核源代码 Documentation/oops-tracing.txt 中或 ksymoops 手册页上有完整的说明可以参考。

将 oops 消息复制保存在一个文件中，通过 ksymoops 工具转换它。

```
$ ksymoops –m System.map < oops.txt
```

这样 oops 信息就转换成符号信息，打印到控制台上了。如果想把结果保存下来，可以把结果重定向到文件中。

（3）内核 kallsyms 选项支持符号信息

Linux 2.6 内核引入了 kallsyms 特性，可以通过定义 CONFIG_KALLSYMS 配置选项启动。该选项可以载入内核映像对应内存地址的符号的名称，内核可以直接跟踪回溯函数名称，而不再打印难懂的机器码了。这样，就不再需要 System.map 和 ksymoops 工具了。因为符号表要编译到内核映像中，所以内核映像会变大，并且符号表永久驻留在内存中，对于开发来说，这也是值得的。

9.4.2 panic

当系统发生严重错误的时候，将调用 panic()函数。

那么 panic 函数执行了哪些操作呢？不妨分析一下 panic 函数的实现。

```
/* kernel/panic.c */
/** panic - 停止系统运行
 *  参数 fmt:  要打印的字符串
 *  显示信息，然后清理现场，不再返回。
 */
NORET_TYPE void panic(const char * fmt, ...)
{
    long i;
    static char buf[1024];
    va_list args;
#if defined(CONFIG_ARCH_S390)
            unsigned long caller = (unsigned long) __builtin_return_address(0);
#endif
      /* 可以不关闭抢占，直接发出 panic 警报。这里调用的函数还是希望关闭抢占。 */
      preempt_disable();
      bust_spinlocks(1);
      va_start(args, fmt);
      vsnprintf(buf, sizeof(buf), fmt, args);
      va_end(args);
      printk(KERN_EMERG "Kernel panic - not syncing: %s\n",buf); /* 打印警报信息 */
      bust_spinlocks(0);
      /* 如果系统已经崩溃，可以调用 crash_kexec 函数处理所有事情 */
      crash_kexec(NULL);
#ifdef CONFIG_SMP
```

```
	/* smp_send_stop 函数是普通的 smp 关闭函数，在 panic 时可能不能正常工作。 */
	smp_send_stop();
#endif
	notifier_call_chain(&panic_notifier_list, 0, buf);
	if (!panic_blink)
		panic_blink = no_blink;
	if (panic_timeout > 0) {
		/* 显示等待的秒数，直到重起机器。这里也不能使用普通的定时器。 */
		printk(KERN_EMERG "Rebooting in %d seconds..",panic_timeout);
		for (i = 0; i < panic_timeout*1000; ) {
			touch_nmi_watchdog();
			i += panic_blink(i);
			mdelay(1);
			i++;
		}
		/*  这里不能关闭所有的东西，完全的重起。但是尽可能让系统重起。 */
		emergency_restart();
	}
#ifdef __sparc__
	{
		extern int stop_a_enabled;
		/* 确认用户可以敲 Stop-A (L1-A) */
		stop_a_enabled = 1;
		printk(KERN_EMERG "Press Stop-A (L1-A) to return to the boot prom\n");
	}
#endif
#if defined(CONFIG_ARCH_S390)
		disabled_wait(caller);
#endif
	local_irq_enable();
	for (i = 0;;) {
		i += panic_blink(i);
		mdelay(1);
		i++;
	}
}
EXPORT_SYMBOL(panic);	/* 输出 panic 函数公用 */
```

panic()函数首先尽可能把出错信息打印出来，再拉响警报，然后清理现场。这时候大概

系统已经崩溃，等待一段时间让系统重启。

对于开发调试过程来说，可以让 panic 打印更多信息或者调试 panic 函数，从而分析系统出错原因。

9.5 内核源码调试

因为 Linux 内核程序是 GNU GCC 编译的，所以对应地使用 GNU GDB 调试器。Linux 应用程序需要 gdbserver 辅助交叉调试。那么内核源代码调试时，谁来充当 gdbserver 的角色呢？

9.5.1 KGDB 调试内核源代码

KGDB 是 Linux 内核调试的一种机制。它使用远程主机上的 GDB 调试目标板上的 Linux 内核。

准确地说，KGDB 是内核的功能扩展，它在内核中使用插桩（Stub）的机制。内核在启动时等待远程调试器的连接，相当于实现了 gdbserver 的功能。然后，远程主机的调试器 GDB 负责读取内核符号表和源代码，并且建立连接。接下来，就可以在内核源代码中设置断点、检查数据并进行其他操作。

KGDB 的调试模型如图 9.2 所示。

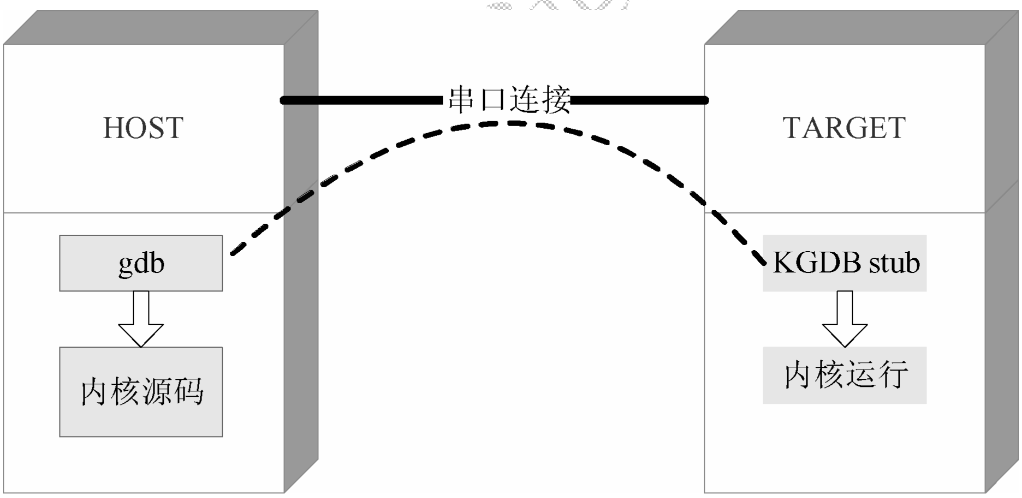


图 9.2 KGDB 调试内核模型

在图 9.2 中，KGDB 调试需要一台开发主机和一台目标板。开发主机和目标板之间通过一条串口线（null 调制解调器电缆）连接。内核源代码在开发机器上编译并且通过 GDB 调试；内核映像下载到目标机上运行。两者之间通过的通信通过串口，Linux 2.6 内核还增加了以太网接口通讯的方式。

遵循下面的步骤来设置 KGDB 调试环境。

（1）配置编译 Linux 内核映像

选择合适的 Linux 内核版本和 KGDB 补丁。配置编译内核，将内核模块静态编译到内核，

因为这是使用 kgdb 最简单的方法。如果动态加载模块，调试起来会麻烦一些。

要让内核进入 KGDB 调试模式，需要在“Kernel hacking”中使能 KGDB 支持的选项。如果找不到这个 KGDB 支持的选项，说明在当前平台上还不支持 LGDB，可以搜索补丁。

还要在 GCC 编译器的选项中添加“-g-ggdb”，使内核映像信息包含调试信息。这可以通过修改顶层 Makefile 中 CFLAGS 变量实现。

另外，还需要配置内核命令行参数，例如：“kgdb=ttyS0,115200n8”。要想调试内核启动阶段代码，可以配置成“kgdb=halt”。

（2）在目标板上启动内核

把编译好的内核映像下载到目标板上，通过 Bootloader 引导启动。

多数 Bootloader 能够向内核传递内核命令行参数。这样不需要因为每次改变内核命令行参数而重新编译内核了。

内核启动解压后，等待远程 gdb 连接，显示下列一行信息。

```
Waiting for connection from remote gdb...
```

（3）启动 gdb，建立连接

创建一个 gdb 启动脚本文件，名字为.gdbinit，保存在内核源文件目录中。脚本.gdbinit 内容如下。

```
#.gdbinit
sh echo -e "\003" > /dev/ttyS0
set remotebaud 115200
symbol-file vmlinux
target remote /dev/ttyS0
set output-radix 16
```

到内核源代码树顶层目录下，启动交叉工具链的 gdb 工具。.gdbinit 脚本将在 gdb 启动过程中自动执行。

如果一切正常，目标板连接成功，进入调试模式。常见的情况是连接不成功，可能是因为串口设置或者连接不正确。

（4）使用 gdb 的调试命令设置断点，跟踪调试

找到内核源代码适当的函数位置，设置断点，继续执行。这样就可以进行内核源代码的调试了。

9.5.2 BDI2000 调试内核源代码

除了软件调试以外，还有硬件工具可以用来进行嵌入式软件调试。尽管硬件工具一般比较贵，但是硬件调试会比软件调试更有效率。对于嵌入式系统开发来说，硬件工具是必要的。

最基本的工具是示波器。它可以用来测量“中断延迟时间”。还可以有其他用途，例如：观察目标板和外界的交互以及电路板上的信号。

不过，示波器并不适合用来分析许多信号同时传输的状态，例如：系统的内存或者 I/O 总线。要分析此类信号，需要使用逻辑分析仪。它可以检查通过总线传送的各种值。

对于操作系统软件的问题，一般还需要仿真器（ICE，In-Circuit-Emulator）或者 BDM/JTAG

调试器来解决。这类工具可以调试非信号层次的软件问题。ICE 依靠拦截处理器和系统其他组件的交互，BDM/JTAG 依靠实现在处理器芯片上的调试接口。出于各种原因，ICE 有逐渐被 BDM 或者 JTAG 调试器取代的趋势。Linux 内核通常是在 BDM 和 JTAG 调试器的协助下移植到新体系结构的。如果准备建立自己的嵌入式系统，电路板应该考虑为开发者提供 BDM 或者 JTAG 接口，这样才能够使用调试器。

在所有的 BDM/JTAG 调试器产品中，Abatron BDI2000 是性价比最高的一种仿真器。BDI2000 可以通过 BDM/JTAG 接口控制处理器，初始化硬件板，通过 10M 以太网接口与开发主机连接。它相当于控制目标板运行的 gdbserver，可以与 gdb 建立远程连接调试。

BDI2000 可以调试运行在内核空间的任何代码，但是不能调试用户空间的应用程序代码。这是因为 BDI2000 可以支持 MMU 功能，它只能工作在连续的地址空间。内核模式代码就存在于一段连续的内存映射上，与其他用户空间的进程内存映射不连续。

Montavista 软件公司使用 BDI2000 调试内核启动和设备驱动程序就是一个很好的例子。下面说明一下 BDI2000 调试 Linux 内核的操作步骤。

（1）主机/目标机设置

主机兼容 PC，运行 Redhat Linux 9。目标机 S3C2410 的开发板，运行 U-Boot。BDI2000 的固件版本说明如表 9.3 所示。

表 9.3　BDI2000 的固件版本说明

组　件	版　本	组　件	版　本
Bdisetup	1.05	Firmware	1.10 bdiGDB for ARM
Loader	1.04	Logic	1.02 ARM

Abatron 的固件需要根据处理器类型和调试器类型更新，可以支持在 Windows 下使用，也可以支持在 Linux 下使用。具体参考 BDI2000 的使用手册。

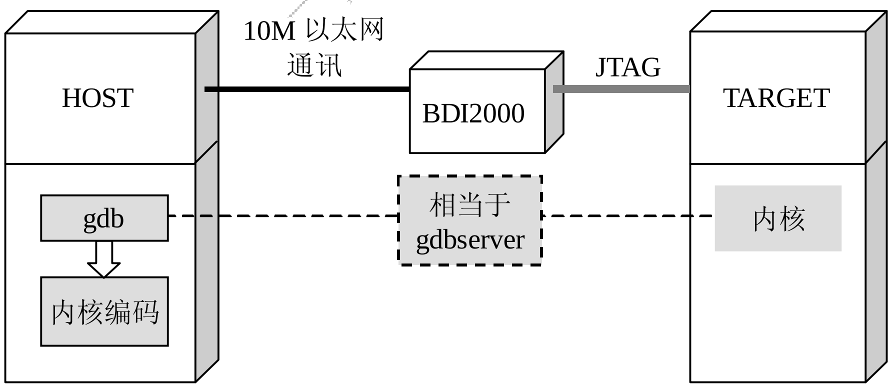


图 9.3　BDI2000 调试连接图

（2）准备要调试的内核

要调试 Linux 内核映像，首先要编译一个带调试符号的内核映像。这需要修改内核源码顶层目录下的 Makefile。这个 Makefile 负责内核的配置和编译。

为了添加内核调试符号信息，需要修改 Makefile 的 CFLAGS，主要不是 HOSTCFLAGS。

使用编辑器打开顶层目录的 Makefile，在 CFLAGS 宏定义末尾添加“-g–ggdb”，然后重新配置编译内核，把编译得到的内核映像复制到 tftp 文件输出目录下。

```
cp arch/arm/boot/zImage /tftpboot/zImage.bdi
```

注意 使用 KGDB 和 BDI2000 的调试方法是互斥的，所以不要配置 KGDB 内核选项。

（3）通过 BDI2000 控制硬件开发板

BDI2000 需要配置文件初始化硬件开发板。这里有两种方法。第 1 种方法是为开发板写一个足够复杂全面的配置文件，可以执行全部初始化；第 2 种方法很简单，只要 BDI2000 能够跳转到开发板固件的位置，执行固件完成初始化。

目标板和 BDI2000 上电。Abatron 建议 BDI2000 先上电，目标板后上电。

在主机上通过 telnet 连接 BDI2000。假设已经在主机的/etc/hosts 文件中配置了 bdi 为对应的 IP 地址。

```
$ telnet bdi
```

telnet 控制台可以看作 BDI2000 的命令窗口。在 BDI>命令提示符下提供了一系列仿真命令。

下面的命令中，凡是 BDI2000 命令窗口的都带 BDI>前缀；GDB 窗口的命令都带“（gdb）”提示符；目标板 Linux 命令带“#”提示符，主机端 Linux 命令用“$”提示符。

（4）设置 BDI2000 断点

建议看明白 Abatron 文档有关 gdb 调试和 MMU 设置的部分。

Abatron 文档特别说明不要在 MMU 还没有打开的代码设置断点，建议在内核初始化使能 MMU 之后设置断点，这样更容易与主机调试器连接。

这些警告意味着：如果不得不从第一条指令单步调试，注意在单步执行打开 MMU 的代码的过程中，调试可能发生的事情。这里假定不需要调试内核初始化最前面的第一条指令，在 MMU 完全打开之后跟踪内核调试。

Start_kernel 函数入口是内核启动过程中设置断点的理想位置。下一步就可以在熟悉的内核或者驱动程序文件中设置断点。编译内核的时候不压缩的内核映像 vmlinux 和符号表 System.map 都生成在源码顶层目录下。这个符号表中的地址都是打开 MMU 时的，使用 gdb 调试的时候需要这个文件。

```
$ grep start_kernel System.map
```

得到 start_kernel 函数的地址 0xXXXXXXXX，然后在 BDI2000 控制台下设置断点。

```
BDI>bi 0xXXXXXXXX
```

（5）下载内核

目标板固件可以用来初始化硬件。为了看到固件已经启动，和 Linux 的控制台一样，需要使用开发主机端的 minicom 与目标机通过串口建立连接。这需要配置 minicom 的波特率等。

```
$ minicom
```

在 BDI2000 控制台下让目标板运行。

```
BDI>go
```

在 minicom 窗口下应该可以看到启动信息和命令提示符。这时把目标板停住。

```
BDI>halt
```

现在 BDI2000 已经控制了目标板，并且完全通过板上固件初始化。可以下载一个内核到目标板上了。

配置文件[HOST]部分举例说明如下。

```
[HOST]
IP 192.168.1.100
FILE   /tftpboot/zImage.bdi
FORMAT BIN 0x30008000
START   0x30008000
LOAD MANUAL
```

通过这些设置，执行 load 下载命令的时候，就可以按照要求把相应的文件下载到相应的地址。

```
BDI>load
BDI>go
```

在 minicom 控制台下面，应该可以看到显示内核解压缩信息。解压缩完毕，都会显示：

```
Now booting the kernel
```

当执行到断点的时候，BDI2000 命令行窗口打印一条消息：

```
-TARGET : target has entered debug mode
```

这时可以通过 info 命令确定当前状态，例如执行 info 命令打印出下列信息，说明进入调试模式的原因。

```
Target state : debug mode
Debug entry cause : instruction breakpoint
Current PC : 0xC0008580
```

当前 PC 指针正好是 System.map 中 start_kernel 的函数入口。

（6）gdb 连接 BDI2000

这里目标板处于停止状态，在 Linux 内核启动过程中，处于 BDI2000 的控制之下。下一步是在开发主机上启动调试器，与目标板建立连接。假定下列的命令在内核源码树的顶层目录下，要调试的文件为 vmlinux。在编译生成 zImage 之前，一定会生成 vmlinux 和它对应的 System.map 文件。添加了调试信息的 vmlinux 文件一般很大。

```
$ arm-linux-gdb vmlinux
```

启动调试器 gdb 的命令。如果喜欢用 gdb 的图形化界面，可以使用 ddd。执行命令如下。

```
$ ddd --debugger arm-linux-gdb –gdb vmlinux
```

gdb 与 BDI2000 的连接仍然是通过 gdb 控制台命令完成的，先确保连接能够成功，然后再使用 ddd。使用 KGDB 调试内核的时候，不建议使用 ddd，因为 KGDB 使用串口通信，gdb 支持图形化接口有额外的花销。

```
(gdb)target remote bdi:2001
```

可以看到连接成功的信息。

（7）设置 gdb 断点

现在就可以通过 BDI2000 在主机上的 gdb 调试器控制内核启动了。最好设置这样 2 个断点，一个是 panic，另一个是 sys_sync。调用 panic 函数都是系统出了严重错误，这里设置断点是很容易理解的。设置另外一个 sys_sync 断点相当于留后门，在登录进入目标板 Linux 系统后，可以再进入调试模式。

```
(gdb)b panic
(gdb)b sys_sync
(gdb)cont
```

cont 命令可以让内核完全启动。如果要对于特定的启动顺序感兴趣，可以预先设置其他启动过程的断点。

上述启动 gdb 并且连接调试的命令，可以通过配置启动文件.gdbinit 自动完成。启动文件可以放在用户的目录下或者当前目录下，当前目录下的文件优先。下面举例说明.gdbinit 文件。

```
# .gdbinit
target remote bdi:2001
b panic
b sys_sync
cont
```

（8）重新控制调试过程

假设在 sys_sync 函数设置了断点，只要在目标机敲 sync 命令，主机端的 gdb 可以在任何时候都可以进入调试模式。当内核完全启动并且登录进入 Shell 之后，可以敲 sync 验证一下。

（9）调试内核模块

BDI2000 真正让 Linux 开发者喜欢的是它调试内核模块的能力，它可以在内核启动之后调试动态加载的内核模块。这种功能对于设备驱动程序开发者非常有用，以免开发过程中每次修改都要重启系统。

“黑色经典”系列之《嵌入式 Linux 系统开发技术详解——基于 ARM》

第 10 章 制作 Linux 根文件系统

本章目标

本章介绍了 Linux 根文件系统的组织结构，并且分析了 init 进程调用文件系统脚本初始化的过程。只有掌握了文件系统的基本构成，才能自己动手定制 Linux 文件系统。

- 根文件系统组织结构
- INIT 系统初始化过程
- 定制文件系统

Linux 的根文件系统具有非常独特的特点，就其基本组成来说，Linux 的根文件系统应该包括支持 Linux 系统正常运行的基本内容，包含着系统使用的软件和库，以及所有用来为用户提供支持架构和用户使用的应用软件。因此，至少应包括以下几项内容。

1．基本的文件系统结构，包含一些必需的目录比如：/dev，/proc，/bin，/etc，/lib，/usr，/tmp 等。

2．基本程序运行所需的库函数，如 Glibc/uC-libc。

3．基本的系统配置文件，比如 rc，inittab 等脚本文件。

4．必要的设备文件支持：/dev/hd*，/dev/tty*，/dev/fd0。

5．基本的应用程序，如 sh，ls，cp，mv 等。

以下章节的内容将对制作 Linux 根文件系统的过程作一些详细地分析，目的是使读者能够较快地理解如何在一个目标系统（Target）建立起操作系统的根文件系统，进而加快开发流程。

10.1 根文件系统目录结构

文件系统是在任何操作系统中都非常重要的概念，简单地讲，文件系统是操作系统用于明确磁盘或分区上的文件的方法和数据结构，即在磁盘上组织文件的方法。文件系统的存在，使得数据可以被有效而透明地存取访问。

进行嵌入式开发，采用 Linux 作为嵌入式操作系统必须要对 Linux 文件系统结构有一定的了解。每个操作系统都有一种把数据保存为文件和目录的方法，因此它才能得知添加、修改之类的改变。在 DOS 操作系统之下，每个磁盘或磁盘分区由独立的根目录，并且用唯一的驱动器标识符来表示，如：C:\, D:\等。不同磁盘或不同的磁盘分区中，目录结构的根目录是各自独立的。而 Linux 的文件系统组织和 DOS 操作系统不同，它的文件系统是一个整体，所有的文件系统结合成一个完整的统一体，组织到一个树形目录结构之中，目录是树的枝干，这些目录可能会包含其他目录，或是其他目录的“父目录”，目录树的顶端是一个单独的根目录，用/表示。在 Linux 下可以看到系统的根目录组成内容，如图 10.1 所示。



图 10.1 Linux 下根目录内容

在上图中，弧线内部的部分即为 Linux 根目录的组成。

10.1.1 FHS 目录结构

Linux 遵守文件系统科学分类标准（Filesystem Hierarchy Standard，FHS），一个定义许多文件和目录的名字和位置的标准，该项标准可以在 http://www.pathname.com/FHS 找到，FHS 也是用来组织 Linux 和 Unix 文件的方法，它使得 Linux 文件系统布局实现了标准化，一个 Linux 的根文件系统目录结构如图 10.2 所示。

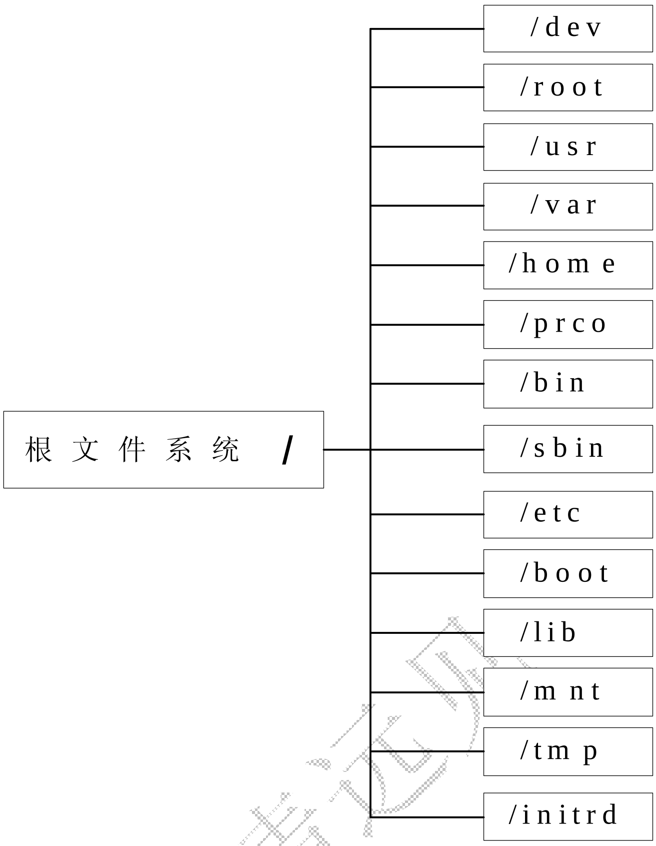


图 10.2 Linux 根文件系统结构

1. /dev 设备文件

在/dev 目录下是一些称为设备文件的特殊文件，用于访问系统资源或设备，如软盘，硬盘，系统内存等。设备文件的概念是 DOS 和 Windows 操作系统中所没有的，在 Linux 下，所有设备都被抽象成了文件，有了这些文件，用户可以像访问普通文件一样方便地访问系统中的物理设备。例如：你可以像从一个文件中读取数据一样，通过读取/dev/mouse 文件从鼠标读取输入信息。在/dev 目录下，每个文件都可以用 mknod 命令建立，各种设备所对应的特殊文件以一定规则来命名。以下是/dev 目录下的一些主要设备文件。

（1）/dev/console

系统控制台，也就是直接和系统连接的监视器。

（2）/dev/hd

在 Linux 系统中，对于 IDE 接口的整块硬盘表示为/dev/hd[a-z]，对于硬盘的不同分区，表示方法为/dev/hd[a-z]n，其中 n 表示的是该硬盘的不同分区情况。例如/dev/hda 指的是第一个硬盘，hda1 则是指/dev/hda 的第一个分区。如系统中有其他的硬盘，则依次为/dev/hdb、/dev/hdc 等；如有多个分区则依次为 hda1、hda2 等。

（3）dev/fd

软驱设备文件。通过前面对系统 IDE 接口硬盘的表示方法不难理解：/dev/fd0 是指系统的第一个软驱，也就是通常所说的 A 盘，/dev/fd1 是指系统的第二个软驱。

（4）dev/sd

SCSI 接口磁盘驱动器。理解方法和 IDE 接口的硬盘相同，只是把 hd 换成 sd。目前，Linux 下驱动 USB 存储设备的方法采用模拟 SCSI 设备，所以 USB 存储设备的表示方法与 SCSI 接口硬盘的表示方法相同。

（5）dev/tty

设备虚拟控制台。如：/dev/tty1 指的是系统的第一个虚拟控制台，/dev/tty2 则是系统的第二个虚拟控制台。

（6）dev/ttyS*

串口设备文件。dev/ttyS0 是串口 1，dev/ttyS1 是串口 2。

2. /root root 用户主目录

root 目录中的内容包括：引导系统的必备文件、文件系统的挂装信息、设备特殊文件、以及系统修复工具和备份工具等。由于是系统管理员的主目录，普通用户没有访问权限。

3. /usr

/usr 是最庞大的目录，该目录中包含了一般不需要修改的命令程序文件、程序库、手册和其他文档等。Linux 内核的源代码就放在/usr/src/linux 里。

4. /var

该目录中包含经常变化的文件，例如打印机、邮件、新闻等的脱机目录、日志文件以及临时文件等。因为该文件系统的内容经常变化，因此如果和其他文件系统，如/usr 放在同一硬盘分区，文件系统的频繁变化将会提高整个文件系统的碎片化程度。

5. /home

用户主目录的默认位置。例如，一个名为 LY 的用户主目录将是/home/LY，系统的所有用户的数据保存在其主目录下。

6. /proc

需要注意的是，/prco 文件系统并不保存在系统的硬盘中，操作系统在内存中创建这一文件系统目录，是虚拟的目录，即系统内存的映射，其中包含一些和系统相关的信息，例如 CPU 的信息等。

7. /bin

该目录包含二进制（binary）文件的可执行程序，这里的 bin 本身就是 binary 的缩写，许多 Linux 命令就是放在该目录下的可执行程序，例如 ls、mkdir、tar 等命令。

8．/sbin

与 bin 目录类似，存放系统编译后的可执行文件、命令，如常用到的 fsck、lsusb 等指令，通常只有 root 用户才有运行的权限。

9．/etc

/etc 目录在 Linux 文件系统中是一个很重要的目录，Linux 的很多系统配置文件就在该目录下，例如系统初始化文件/etc/rc 等。Linux 正是靠这些文件才得以正常地运行，用户可以根据实际需要来配置相应的配置文件，以下列举一些配置文件。

（1）/etc/rc 或/etc/rc.d

启动或改变运行级别时运行的脚本或脚本的目录。大多数的 Linux 发行版本中，启动脚本位于/etc/rc.d/init.d 中，系统最先运行的服务是那些放在/etc/rc.d 目录下的文件，而运行级别在文件/etc/inittab 里指定，这些会在后面的内容中详细讲到。

（2）/etc/passwd

/etc/passwd 是存放用户的基本信息的口令文件。该口令文件的每一行都包含由 6 个冒号分隔的 7 个域，其中的域给出了用户名、真实姓名、用户起始目录、加密口令和用户的其他信息。

- username：用户名。
- passwd：是口令密文域。密文是加密过的口令。如果口令经过 shadow 则口令密文域只显示一个 x，通常，口令都应该经过 shadow 以确保安全。如果口令密文域显示为*，则表明该用户名有效但不能登录。如果口令密文域为空则表明该用户登录不需要口令。
- uid：系统用于唯一标识用户名的数字。
- gid：表示用户所在默认组号。
- comments：用户的个人信息。
- directory：定义用户的初始工作目录。
- Shell：指定用户登录到系统后启动的外壳程序。

（3）etc/fstab

指定启动时需要自动安装的文件系统列表。通常来讲，如果用户在使用过程中需要手动加载许多文件系统，这会带来不小的工作量。为了避免这样的麻烦，让系统在启动的时候自动加载这些文件系统，Linux 中使用/etc/fstab 文件来完成这一功能。fstab 文件中列出了引导时需安装的文件系统的类型、加载点及可选参数。所以进行相应的配置即可确定系统引导时加载的文件系统。

（4）etc/inittab

init 的配置文件，在后面的内容会详细讲到。

10．/boot

该目录存放系统启动时所需的各种文件，如内核的镜像文件，引导加载器（bootstrap loader）使用的文件 LILO 和 GRUB。

11．/lib

标准程序设计库，又叫动态链接共享库，作用类似于 Windows 里的.dll 文件。

12．/mnt

该目录用来为其他文件系统提供安装点，例如可以在该目下新建一目录 floppy 用来挂载软盘，同样可以新建一目录 cdrom（可以用任意名称）用来挂载光盘等。比如在 Linux 下的终端执行下面的语句：

```
# mount –t  vfat  dev/hda1  /mnt/win_D
```

即可将硬盘的第一个分区挂载到 Linux 下的/mnt/win_D 目录中。

13．/tmp 公用的临时文件存储点

14．/initrd

用来在计算机启动时挂载 initrd.img 映像文件以及载入所需设备模块的目录，需要注意的是，不要随便删除/initrd/目录，如果删除了该目录，将无法重新引导系统。

10.1.2 文件存放规则

为了实现各种 Linux 版本系统的标准化，各种不同的 Linux 版本都会根据 FHS（Filesystem Hierarchy Standard）标准来进行系统管理，这也使得 Linux 系统的兼容性大大提高。FHS 规定了两级目录，第一级是根目录下的主要目录，根据目录名称可以得知其中应该放置什么样的文件，比如/etc 应该放置各种配置文件，/bin 和/sbin 目录下应该放置相应的可执行文件等；第二级目录则主要针对/usr 和/var 做出了更深层目录的定义。

Unix/Linux 系统很长时间以来一直是在“什么文件放在哪里”的基础之上建立文件存放规则的，并且按照这些规则把文件放进相应分级结构里。文件系统分级结构标准（FHS）试图以一种合乎逻辑的方式定义这些规则，而且在 Linux 上得到了广泛应用。按照 FHS 标准，在 Linux 下存放文件主要有以下的一些规则。

1．把全局配置文件放入/etc 目录下。

2．将设备文件信息放入/dev 目录下，设备名可以作为符号链接定位在/dev 中或/dev 子目录中的其他设备存在。

3．操作系统核心定位在/或/boot，若操作系统核心不是作为文件系统的一个文件存在，不应用它。

4．库存放的目录是/lib。

5．存放系统编译后的可执行文件、命令的目录是/bin，/sbin，/usr。

10.2 添加系统文件

10.2.1 添加共享链接库

Linux 下的共享函数库在根文件系统下的/lib 目录中，应用程序通常都是需要共享库才可以工作的，所以添加共享库对于根文件系统是必需的。图 10.3 是一个 Linux 系统下/lib 目录下的内容。

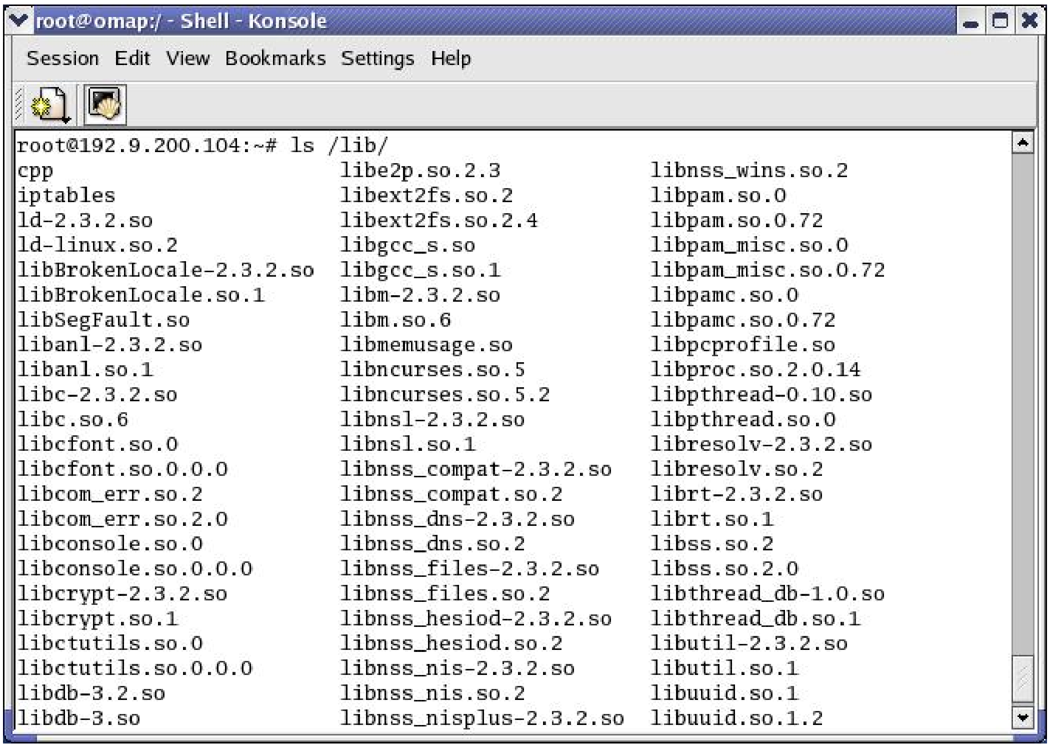


图 10.3 Linux 下/lib 目录的内容

每个 Linux 系统或嵌入式 Linux 系统都需要一个 C 库。C 库提供了常用的文件操作（比如打开、读/写）、内存管理操作（malloc 和 free）等其他的一些函数。许多基于 X86 架构的 Linux 系统使用 Glibc 库。但是对于嵌入式系统开发来说，使用 Glibc 对于内存的消耗也较多，如果内存资源大小受到严格限制的话，采用 Glibc 是不可接受的。uClibc 是针对嵌入式系统开发的，这是一个稳定的、兼容 Glibc 的替代品，所以它力图成为完整但结构紧凑的 C 库。在绝大多数情况下，针对 uClibc 编译的应用程序和工具与针对 Glibc 编译的没有区别。

在根文件系统的/lib 目录下主要包含以下 4 种类型的文件。

1. 实际的共享链接库

这类文件名的格式为 lib libname version.so，其中，libname 是共享库的名称，version 是版本编号。例如：glibc 2.2.3 的数学链接库的名称为 libm-2.2.3.so。

2. 主修订版本的符号链接

主修订版本的符号链接的格式为 lib libname.so.major-revision-version，例如，C 链接库的符号链接的名称为 libc.so.6。程序一旦链接了特定的链接库，它将会参用其符号链接，程序启动时，加载器在加载程序之前，会因此加载该文件。

3. 与版本无关的符号链接指向主修订版本的符号链接

这些符号链接的主要功能是为需要链接特定链接库的所有程序提供一个通用的条目，与主修订版本的编号或 glibc 涉及的版本无关。这些符号链接的格式为 lib libname.so。

4. 静态的链接库

选择以静态方式链接链接库的应用程序会使用这些文件，这些文件的格式为：lib libname.a。

在 Linux 程序开发过程当中，应用程序的执行离不开共享链接库的支持，所以需要将其中的一些文件复制到用户目标板的根文件系统的相应位置。事实上，需要的文件是共享链接库和主修订版本的符号链接。

为了明确用户应用程序需要链接哪些链接库，通常可以使用系统下的命令 lddl 来列出应用程序要以存哪些动态链接库。例如查看文件复制命令 cp 所依赖的共享库，可以执行如下指令。

```
# ldd /bin/cp
libtacl.so.1 => /lib/libacl.so.1 (0x00701000)
libselinux.so.1 => /lib/libselinux.so.1(0x00b87000)
libc.so.6 => /lib/tls/libc.so.6 (0x0064b000)
libattr.so.1 => /lib/libattr.so.1(0x00ba1000)
/lib/ld-linux.so.2 (0x40000000)
```

“=>”左边的表示该程序所需共享库的符号链接名称，右边表示由 Linux 的共享库系统找到的对应的共享库在根文件系统中的实际位置，所以可看到执行 cp 指令需要用到 5 个共享库。默认情况下，动态链接库的配置文件/etc/ld.so.conf 中包含有默认的共享库搜索路径，通过查看改配置文件来了解默认的共享库搜索路径。例如：

```
#vi /etc/ld.so.conf
/usr/X11R6/lib
/usr/lib
/usr/local/lib
/usr/lib/qt-3.3/lib
```

以上是笔者 Fedora Core Linux 下的动态链接库的配置文件的内容。通常情况下，许多开放源代码的程序或函数库都会默认安装到/usr/local 目录下的相应位置（/usr/local/bin 或/usr/local/lib），以便与系统自身的程序或函数库相区别。而许多 Linux 系统的/etc/ld.so.conf 文件中默认又不包含/usr/local/lib。因此，往往会出现已经安装了共享库，但是却无法找到共享库的情况。这时，就应该检查/etc/ld.so.conf 文件，如果其中缺少/usr/local/lib 目录，就应该添加进去。

如同 Glibc 一样，uClibc 也包含了若干链接库，因为 uClibc 是 Glibc 的替代品，所以 uClibc 中各个组件的名称及其用法如同 glibc 中相应的组件，但是 uClibc 并不实现 Glibc 的所有组件，uClibc 只会实现 ld、libc、libcrypt、libdl、libm、libpthread、libresolv 和 libutil。决定好所需的组件清单之后，接着可以将它们及相关的符号链接复制到目标板的根文件系统中的/lib 目录中，如下面的命令会将 uClibc 的所有组件复制到目标系统的根文件系统。

```
# cd  /lib                  ;主机的根文件系统的/lib 目录
# cp *-*.so      /rootfs/lib
# cp -d *.so.[*0-9] /rootfs/lib
```

第一个 cp 复制命令会复制实际的共享链接库，第二个 cp 复制命令会复制主修订版本的符号链接。当把 uClibc 组件安装到目标系统的根文件系统中以后，应用程序在执行的过程中就可以使用这些共享库了。

10.2.2 添加内核模块

接下来了解 Linux 系统添加内核模块的过程。首先来阐述内核模块的概念，简单地讲，内核模块是一些可以让系统内核在需要时加载并且能够执行；在不需要的时候可以被系统卸载掉的代码。添加内核模块是嵌入式 Linux 开发中非常有用而又很重要的一项环节，在嵌入式系统开发过程当中，如果想增加系统的某部分功能，可以有 2 种方法：一种方法是编译内核：即把相应部分在编译内核时候编译进去，另一种方法就是采用动态加载，即动态调用系统所需要的内核模块。

采用以上的两种方法各有优缺点，如果编译到内核中，在内核启动时就可以自动支持相应部分的功能，这样的优点是方便、速度快，机器一启动，你就可以使用这部分功能了；缺点是会使内核变得庞大起来，不管你是否需要这部分功能，它都会存在，对于经常用到的部分直接可以考虑直接编译到内核中，比如网卡。如果采用后一种方法也就是编译成模块，就会生成对应的.o 文件，在使用的时候可以动态加载，优点是不会使内核过分庞大，缺点是开发者得自己来调用这些模块。下面就这两种方法分别介绍为系统添加内核模块的过程。

1. 在内核编译过程中自动添加内核模块

内核编译的详细过程会在本书的其他章节中做出详细地介绍，其中对于模块支持的设置选项中包含有 3 项内容。

```
——>Loadable module support
[ * ] Enable loadable module support
[  ] Set version information on all module symbols
```

```
[ ] Kernel module loader
```

第 1 项的内容是指是否支持动态加载内核模块，如果不是所有需要的内容都编译到内核里，应该选择该项；第 2 项的内容可以不选；第 3 项的内容是指让内核在启动时就可以加载所需的模块，这一选项应该选上。在配置内核相关选项之后，对于模块的管理还需执行如下的指令。

```
# make modules
# make modules_install
```

make modules 和 make modules_instal 命令分别生成相应的模块和把模块拷贝到需要的目录中，具体的解释可以参考本书内核编译的相关章节。内核编译之后，就可以将编译内核过程中生成的内核模块复制到目标系统的根文件系统，接着还需要为目标开发系统添加内核模块的配置文件/etc/modprobe.conf，以便系统在运行的时候可以自动加载内核模块，图 10.4 所示就是 Fedora Core 4 下的/etc/modprob.conf 文件。



图 10.4 /etc/modprob.conf 文件

开机自动挂载模块位于该配置文件中，在该文件中，写入了模块的加载命令或模块的别名的定义等；如果想让一些模块开机自动加载，就可以在该配置文件中写入。比如在 modprobe.conf 配置文件中下面的语句。

```
alias eth0 8139too
```

这样系统启动的时候，首先会加载 8139too 模块，同时指定网络设备 8139too 的别名为 eth0。

2．动态添加内核模块

Linux 为了不需要重新编译内核就可以动态加载内核模块，引入了可加载内核模块 LKM（Loadable Kernel Modules）的概念，模块不被编译在内核中，LKM 扩展了操作系统内核的功能而不需要重新编译内核，如果想在 Linux 下查看内核已经加载的内核模块，可以通过执行 lsmod 命令来查看（读取/proc/modules 文件获取所需信息），如图 10.5 所示。

在图 10.5 中，module 指的是模块名称，Size 指的是该模块所占内存页面的大小，而 Used

by 指的是该模块被系统调用的次数。

动态加载内核模块有 2 种方法，以下分别叙述。

（1）采用 modprobe 命令加载

modprobe 常用的功能就是挂载模块，在挂载某个内核模块的同时，这个模块所依赖的模块也被同时挂载，modprob 指令的格式如下。

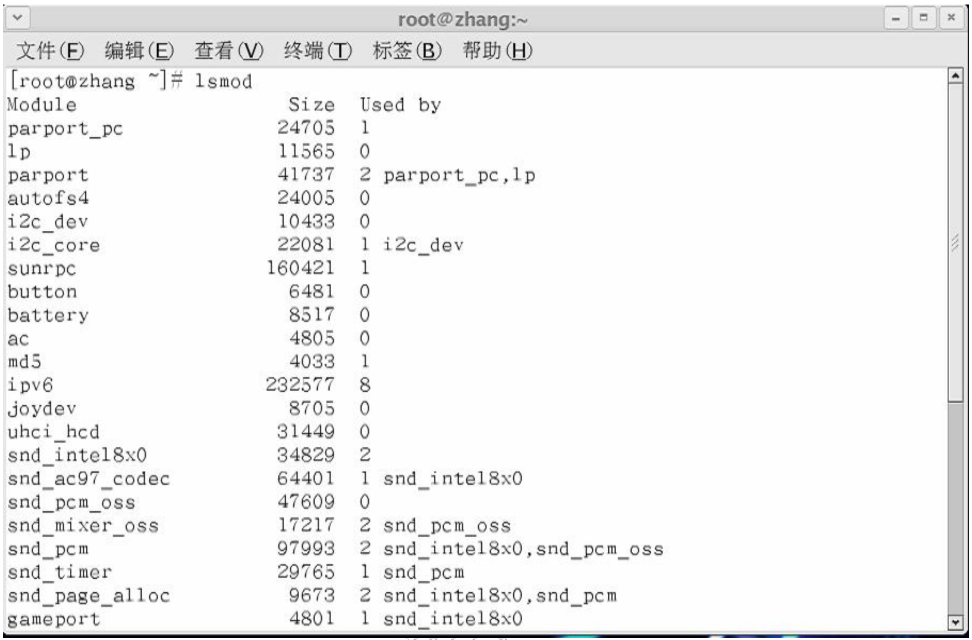


图 10.5　lsmod 命令查看系统加载的内核模块

```
    modprobe [-v] [-V] [-C config-file] [-n] [-i] [-q] [-o <modname>] <modname>
[parameters...]
    modprobe -r [-n] [-i] [-v] <modulename> ...
    modprobe -l -t <dirname> [ -a <modulename> ...]
```

例如：

```
    [root@localhost zhang]# modprobe  8139too      #挂载 8139too 模块;
    [root@localhost zhang]# modprobe   vfat        #挂载 vfat 模块
```

注意　这里的模块名是不带有后缀的。

（2）采用 insmod 命令加载

insmod 指令和 modprobe 指令在功能上有所区别，modprobe 在加载模块时不用指定模块文件的绝对路径，也不用带模块文件的后缀.o 或.ko；而 insmod 需要的是模块的所在目录的绝对路径，并且一定要带有模块文件名后缀的（modulefile.o 或 modulesfile.ko）。例如：

```
    # insmod   /lib/modules/2.6.9-11.EL/Kernel/drivers/pci/hotplug/capiphp.ko
```

注意 采用该种方法添加内核模块要有绝对路径，同时要有完整文件名的后缀。

10.2.3 添加设备文件

Linux 中任何对象（包括设备）都可以认为是文件。Linux 将设备分为最基本的 2 大类：一类是字符设备（Character Device），另一类是块设备（Block Device）。字符设备特殊文件进行 I/O 操作不经过操作系统的缓冲区，而块设备特殊文件用来同外设进行定长的包传输。字符特殊文件与外设进行 I/O 操作时每次只传输一个字符。而对于块设备特殊文件来说，在外设和内存之间一次可以传送一整块数据。在 Linux 根文件系统中，设备文件所在的目录是/dev，图 10.6 显示了 Fedora Core 4.0 linux 目录/dev 中的内容。

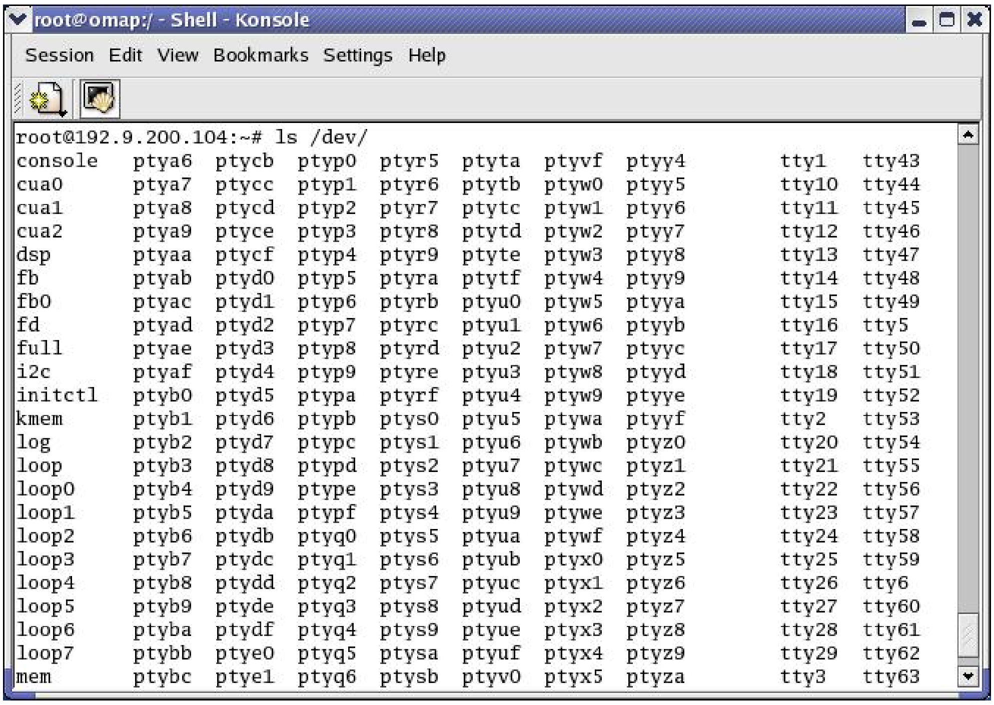


图 10.6 Linux 下/dev 目录中的内容

可以看到/dev 目录下有非常多的设备文件，事实上，嵌入式 Linux 是个定制的系统，目标系统的设备文件只要满足实际开发需求即可，所以在/dev 目录下只添加必要的设备文件即可。Linux 下添加设备文件可以采用以下的方法。

1．使用 mknod 指令来添加设备

举例说明在根文件系统下使用 mknod 指令添加设备文件的过程。

```
#cd  /dev
#mknod  -m  666  null     c  1  3
#mknod  -m  666  zero     c  1  5
```

```
#mknod -m 666 /dev/ttyS0 c 4 64
#mknod -m 600 console  c  5  1
```

注意　所有写入设备/dev/null 的信息都将被隐藏。

添加好基本的设备文件之后，在根文件系统的/dev 目录下还必须包括必要的符号链接，可以使用“ln–s 链接名链接目标”命令建立这些链接，比如：

```
# ln –s  /proc/self/fd  fd
# ln –s  fd/0       stdin
# ln -s fd/1 stdout
# ln -s fd/2 stderr
```

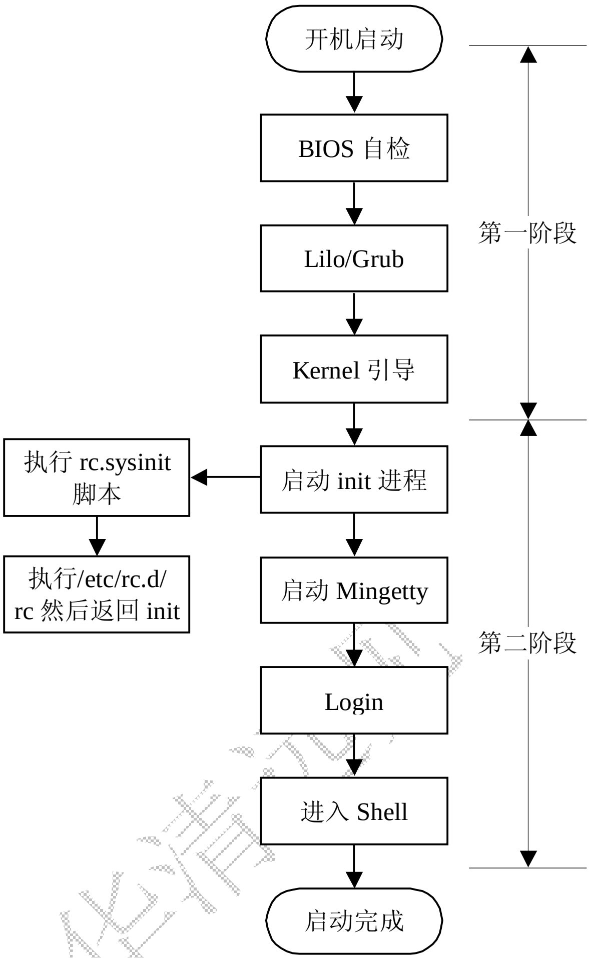
2．在/dev 目录下采用 MAKEDEV（符号链接/sbin/MAKEDEV）来建立设备文件

例如需要在根文件系统中添加 tty0 设备可以输入如下指令。

```
# cd /dev
# ./MAKEDEV ttyS0
```

10.3　init 系统初始化过程

系统的引导和初始化是操作系统实现控制的第一步，是集中体现系统整体性能至关重要部分。了解系统的初始化过程，对于进一步掌握后续开发是十分有帮助的。首先来了解一下 Linux 内核的启动过程，如图 10.7 所示



。

图 10.7 Linux 内核启动过程示意图

通常，Linux 内核的启动可以分为两个阶段。

1．在第 1 个阶段完成硬件检测、初始化和内核的引导；在内核启动的第 1 个阶段，系统按 BIOS 中设置的启动设备（通常是硬盘）启动，接着利用 Lilo/Grub 程序来进行内核的引导工作，内核被解压缩并装入内存后，开始初始化硬件和设备驱动程序。

2．在第 2 个阶段就是 init 的初始化进程。所谓的 init 进程，是一个由内核启动的用户级进程，也是系统上运行的所有其他进程的父进程，它会观察其子进程，并在需要的时候启动、停止、重新启动它们，主要用来完成系统的各项配置。init 从/etc/inittab 获取所有信息。init 程序通常在/sbin 或/bin 下，它负责在系统启动时运行一系列程序和脚本文件，而 init 进程也是所有进程的发起者和控制者。内核启动（内核已经被载入内存，开始运行，并已初始化所有的设备驱动程序和数据结构等）之后，便开始调用 init 程序来进行系统各项配置，也即成为系统的第一个进程，该进程对于 Linux 系统正常工作是十分重要的。

10.3.1 inittab 文件

Linux 启动时，运行一个叫作 init 的程序，然后根据运行级启动后面的任务，包括多用户环境，网络等。所谓的运行级就是操作系统当前正在运行的功能级别。这个级别从 0～6，具有不同的功能。这些级别在/etc/inittab 文件里指定。这个文件是 init 程序寻找的主要文件，init 进程中所做的每一步配置工作都由/etc/initab 中的内容来决定的。以下是 Fedro Core4 的/etc/inittab 文件代码。

```
# inittab      This file describes how the INIT process should set up
#             the system in a certain run-level.
#
# Author:      Miquel van Smoorenburg, <miquels@drinkel.nl.mugnet.org>
#             Modified for RHS Linux by Marc Ewing and Donnie Barnes
# Default runlevel. The runlevels used by RHS are:
#   0 - halt (Do NOT set initdefault to this)
#   1 - Single user mode
#   2 - Multiuser, without NFS (The same as 3, if you do not have networking)
#   3 - Full multiuser mode
#   4 - unused
#   5 - X11
#   6 - reboot (Do NOT set initdefault to this)
#
id:5:initdefault:
# System initialization.
si::sysinit:/etc/rc.d/rc.sysinit
l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6
# Trap CTRL-ALT-DELETE
ca::ctrlaltdel:/sbin/shutdown -t3 -r now
# When our UPS tells us power has failed, assume we have a few minutes
# of power left.  Schedule a shutdown for 2 minutes from now.
# This does, of course, assume you have powerd installed and your
# UPS connected and working correctly.
pf::powerfail:/sbin/shutdown -f -h +2 "Power Failure; System Shutting Down"
```

```
# If power was restored before the shutdown kicked in, cancel it.
pr:12345:powerokwait:/sbin/shutdown -c "Power Restored; Shutdown Cancelled"
# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6
# Run xdm in runlevel 5
x:5:respawn:/etc/X11/prefdm –nodaemon
```

从代码中可以看到，etc/inittab 中语句的每一行包含 4 个域，格式如下。

```
Id: runlevels: action: process
```

（1）id:id 是指入口标识符，它是一个字符串，是由两个独特的字元所组成的标识符号，对于 getty 或 mingetty 等其他 login 程序项，要求 id 与 tty 的编号相同，否则 getty 程序将不能正常工作。

（2）run levels（运行级别）运行级别会指出下一个操作域中的 action 以及 process 域会在哪些 runlevel 中被执行。而在正常的启动程序之后，root 用户可以使用 telinit 这个指令来改变系统的 runlevel。假定在 Linux 系统中 runlevel 的预设值是 5，那么只有那些每一列中 runlevel 域的值为 5 时，后面的 process 才会被执行。所以，如果系统的 runlevel 值不同的话，所执行的 process 也不一样，所以系统启动的资源配置情况在每个不同的 runlevel 下就会有差异。

（3）action:action 域指出的是 init 程序执行相应 process 时，对 process 所采取的动作。比如：只执行 process 一次，还是在它退出时重启。

（4）process 为具体的执行程序。程序后面可以带参数。

现将 etc/inittab 文件代码分析如下。

运行级别定义

```
# 0 - 停机（不要把 initdefault 设置为 0，否则开机之后就会自动关机 ）
# 1 - 单用户模式
# 2 - 多用户模式，但是没有 NFS
# 3 - 完全多用户模式
# 4 - 没有使用
# 5 – X-windows 模式
# 6 – 系统重新启动 （不要把 initdefault 设置为 6,否则开机之后就会重启 ）
```

```
id:5:initdefault:
```

解释：该命令指出缺省的运行级别为 5，即开机后进入 X-window 模式。

```
si::sysinit:/etc/rc.d/rc.sysinit
```

```
l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6
```

解释：系统启动之后自动执行/etc/rc.d/rc.sysinit 脚本，而且指出当运行级别为 5 时，以 5 为参数运行/etc/rc.d/rc 5 脚本，init 进程将等待其返回（wait）。

```
# Trap CTRL-ALT-DELETE
ca::ctrlaltdel:/sbin/shutdown -t3 -r now
```

解释：在启动过程中如果按 Crtl-Alt-Delete，将执行/sbin/下的命令。

```
Shutdown –t3 –r now来重新启动系统。
# When our UPS tells us power has failed, assume we have a few minutes
# of power left.  Schedule a shutdown for 2 minutes from now.
# This does, of course, assume you have powerd installed and your
# UPS connected and working correctly.
pf::powerfail:/sbin/shutdown -f -h +2 "Power Failure; System Shutting Down"
```

解释：如果系统带有 UPS 电源工作，该行命令设定系统在掉电时提示“电源关闭，系统正在关闭”，并且在 2min 后自动关机。

```
# If power was restored before the shutdown kicked in, cancel it.
pr:12345:powerokwait:/sbin/shutdown -c "Power Restored; Shutdown Cancelled"
```

解释：如果工作电源恢复，该命令行提示“电源恢复，取消关机”，并且取消关机。

```
# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6
```

解释：init 进程打开 6 个终端（虚拟控制台），以 tty n 为参数执行/sbin/mingetty 程序，打开 tty n 终端用于用户登录。

```
# Run xdm in runlevel 5
x:5:respawn:/etc/X11/prefdm –nodaemon
```

解释：在级别 5 上运行 xdm 程序，提供 xdm 图形方式登录界面，并在退出时重新执行

（respawn）。

10.3.2 System V init 启动过程

在介绍 System V init 启动过程之前，先来了解一下 System V 的由来。简单地讲，System V 也被称作为 AT&T System V，是 Unix 操作系统众多版本中相当重要的一个。它最初由 AT&T 开发，在 1983 年第一次发布。一共发行了 4 个 System V 的主要版本：版本 1、2、3 和 4。System V Release 4 是其中最成功的版本，它具有一些 Unix 共同的特性，比如 Sys V init 初始化脚本。

概括地讲，Linux \Unix 系统一般有两种不同的初始化启动方式。

- BSD system init
- System V system init

System V 模式将启动档案存放在/etc/...或/etc/rc.d/...及其下的一堆子目录中，主要内容如下。

```
/etc/rc.d/init.d
/etc/rc.d/rc0.d
/etc/rc.d/rc1.d
/etc/rc.d/rc2.d
/etc/rc.d/rc3.d
/etc/rc.d/rc4.d
/etc/rc.d/rc5.d
/etc/rc.d/rc6.d
rc
rc.local
rc.sysinit
```

大多数发行套件的 Linux 都使用了与 System V init 相仿的 init 也就是 Sys V init，它比传统的 BSD system init 更容易使用而且更加灵活，Sys V init 主要思想是定义了不同的“运行级别（runlevel）”。通过配置文件/etc/inittab，定义了系统引导时的运行级别，进入或者切换到一个运行级别时做什么。每个运行级别对应一个子目录/etc/rc.d/rc n.d（n 表示运行级别 0～6），例如，rc0.d 便是 runlevel 0 启动脚本存放的目录，rc3.d 是 runlevel 3，其他依此类推。rc n.d 中的脚本并不是各自独立的，其实它们都通过符号链接连接到/etc/rc.d/init.d 中的脚本。图 10.8 是系统 rc5.d 目录中的内容列表。



图 10.8 /etc/rc.d/rc5.d 内容列表

可以看到里面的内容是一些以字母‘S’和‘K’开头的符号链接，链接指向/etc/rc.d/init.d中的脚本，每个脚本对应一项服务程序。以 S 开头的，表示 Start 启动之意，以 start 为参数调用该脚本；以 K 开头的，则是表示 stop 停止，以 stop 为参数调用该脚本，这就使得 init 可以启动和停止服务。事实上，可以通过手动执行来启动或停止相关服务，比如，可以执行下面的语句分别来启动和停止 NFS 服务。

```
# /etc/rc.d/init.d/nfs      start
启动 NFS 服务               [OK]
# /etc/rc.d/init.d/nfs      stop
停止 NFS 服务               [OK]
```

以下是一个大致的 System V init 过程。

（1）init 过程执行的第一个脚本文件是/etc/rc.d/rc.sysinit，限于篇幅的原因，此处不再列出该脚本文件的详细内容，/etc/rc.d/rc.sysinit 主要做在各个运行级别中进行初始化工作，包括以下内容。

- 启动交换分区。
- 检查磁盘。
- 设置主机名。
- 检查并挂载文件系统。
- 加载并初始化硬件模块。

（2）执行缺省的运行级模式。

这一步的内容主要在/etc/inittab 中体现，inittab 文件会告诉 init 进程要进入什么运行级别，以及在哪里可以找到该运行级别的配置文件。

（3）执行/etc/rc.d/rc.local 脚本文件。

这也是 init 过程中执行的最后一个脚本文件，所以用户可以在这个文件中添加一些需要在登录之前执行的命令，默认地，/etc/rc.d/rc.local 会用使用系统的内核版本和机器类型创建一个登录标志。

（4）执行/bin/login 程序。

login 程序会提示用户输入账号及密码，接着编码并确认密码的正确性，若二者相合，则为使用者进行初始化环境，并将控制权交给 Shell。

10.3.3 Busybox init 启动过程分析

有关 Busybox 的详细介绍会在下一章提到，此处着重介绍 Busybox init 启动过程。与一些标准的 init 比如 Sys v init 一样，Busybox 也具有处理系统初始化过程的能力，由于 Busybox 自身的一些特点，Busybox init 非常适合在嵌入式系统开发中使用，被誉为“嵌入式 Linux 的瑞士军刀”，它可以为嵌入式系统提供主要的 init 功能，通过定制可以做得非常精炼。

默认的情况下，Busybox 安装之后会生成一个可执行程序 Busybox，在目录…/_install/bin 下，查看 Busybox 的属性可以知道/sbin/init 是其符号链接，如果使用 Busybox 做 Ramdisk，BusyBox 会在内核刚完成加载后就立即启动，此后 Busybox 会跳转到它的 init 进程开始执行，它的 init 进程主要进行以下的工作。

- 为 init 进程设置信号处理进程。
- 对控制台进行初始化。
- 解析 inittab 文件即/etc/inittab。
- 在默认情况下，Busybox 会运行系统初始化脚本/etc/init.d/rcS。
- 运行导致 init 暂停的 inttab 命令（动作类型 wait）。
- 运行仅执行一次的 inittab 命令（动作类型 once）。

当 init 进程对控制台进行初始化完成之后，Busybox 会去检查/etc/inittab 文件是否存在，如果存在，就会解析该文件并执行相应的运行级，Busybox 所能够识别的 inittab 文件格式在 Busybox 的安装目录下有详细的说明。

Busybox init 所支持的 inittab 动作类型在 Busybox 安装目录下的 init.c 程序中得到定义。

```
static const struct init_action_type actions[] = {
     {"sysinit", SYSINIT},
     {"respawn", RESPAWN},
     {"askfirst", ASKFIRST},
     {"wait", WAIT},
     {"once", ONCE},
     {"ctrlaltdel", CTRLALTDEL},
     {"shutdown", SHUTDOWN},
     {"restart", RESTART},
     };
```

- Sysinit：为 init 提供初始化命令行的路径。

- Respawn：在相应的进程结束时就重新启动。
- Askfirst：类似于 respawn，主要用途是减少系统上执行的终端应用程序的数量，会在控制台上显示“Please presss Enter to active this console”的信息，并在系统重新启动进程之前等待用户按下“Enter”键。
- Wait：wait 动作会通知 init 必须等到相应的进程执行完之后才能继续执行其他的动作。
- Once：进程只执行一次，而且不会等待他完成。
- Ctrlaltdel：当按下 Ctrl-Alt-Delete 组合键时运行的进程。
- Shutdown：当系统关机时运行的进程。
- Restart：当 init 进程重新启动的时候执行的进程，事实上就是 init 本身。

Busybox 支持的 inittab 文件格式如下所示。

Id：runlevel：action：process

这里面要提出注意的一点是，Busybox 的 init 程序所认识的/etc/inittab 的格式尽管与 Sys V init 非常类似，但其中的操作域 id 具有不同的含义。Busybox 中的 id 用来指定启动的控制台，如果所启动的进程不是可以交互的 Shell，比如 Busybox 的 sh，就可以空着 id 的操作域不用去填写，在为 Busybox 准备 inittab 文件的时候，可以实际参考 Busybox 的安装目录下的 inittab 文件范例，此处不再对该范例文件做详细的解释。

如果 Busybox 没有找到 inittab 文件，Busybox 会使用缺省的配置。

```
#   : : sysinit: /etc/init.d/rcS
/*Busybox init 进程执行的第一个脚本文件*/
#   : : askfirst: /bin/sh
/*在控制台启动一个"askfirst"shell*/
#   : : ctrlaltdel: /sbin/reboot
#   : : shutdown: /sbin/swapoff -a
#   : : shutdown: /bin/umount -a –r
/*系统关机的时候会执行 umount 命令卸载所有的文件系统*/
#   : : restart: /sbin/init
/*等待重新启动 init 进程*/
```

注意 此处的操作域 id 和 runlevel 都是空着的，Busybox 也会忽略 runlevel 操作域。

从以上的分析可以看出，不论 Busybox 是否能找到 inittab 文件，Busybox 下的 init 进程执行的第一个脚本文件都是/etc/init.d/rcS，而不是 Sys V init 结构下执行的脚本文件/etc/rc.d/rc.sysinit，这一点需要注意。

10.4 定制文件系统

10.4.1 定制应用程序

一个嵌入式系统的应用程序，基本上可以分为以下两类。

- 系统基本应用程序，如 ls、cp、mkdir、Telnet 等
- 用户开发的基于特定应用的程序

标准的 Linux 发行版具有功能种类非常多的应用程序，在嵌入式开发过程中，在满足系统基本性能要求的前提下应该适当地精简系统应用程序，这样可以尽可能地减少系统不必要的资源浪费。

在定制嵌入式系统的系统应用程序时，如果把常用的应用程序的源码都下载来交叉编译，这一过程的工作量显然是很大的，而且非常繁琐。为了进一步减小所创建的根文件系统的尺寸，可以考虑使用下列工具包软件来替代某些标准的工具。

事实上使用 Busubox 来定制是一个不错的选择，Busybox 具有 Shell 的功能，它可以提供系统基本的几十个各种应用程序，这其中包括有一个迷你的 vi 编辑器，系统不可或缺的/sbin/init 程序，以及其他诸如 sed，ifconfig，halt，reboot，mkdir，mount，ln，ls，echo，cat...等。此外，Busybox 本身对系统资源的消耗是非常有限的，有关 Busybox 的使用，会在本书相应的章节中有详细的介绍。

在实际存放用户应用程序的时候，一般应遵循 FHS 标准，比如一些基本的二进制应用程序应该存放在/bin 目录下面，当然，也可以根据个人实际情况来处理，比如新增加一个特定目录来存放自己的应用程序。

10.4.2 配置应用程序自动启动

类似于在 Windows 系统下面程序的自动执行一样，在 Linux 下面也可以进行相应的配置实现应用程序的自动启动，在 Linux 下配置应用程序自动启动大致有以下的 3 种方法。

（1）在启动/etc/init.d/下添加启动脚本，创建/etc/rc.d/.../目录下的链接。

事实上，大多数的 Linux 类系统使用/etc/init.d/（或/etc/rc.d/init.d，或其他类似名字的目录）下面的脚本来配置应用程序自动启动。

例如，在某些 Linux 系统上，cron 通过/etc/init.d/cron 脚本启动，Apache 通过/etc/init.d/httpd 启动，syslogd 通过/etc/init.d/syslogd 启动，而 sshd 则通过/etc/init.d/sshd 脚本启动。

一般地，这些脚本通过来自特定 rc.d 目录的符号链接运行。为了配置从哪个 rc.d 目录运行脚本，Linux 系统提供了许多不同的工具，同时也可以手工进行配置。Linux 系统有一个包含所有实际启动脚本文件的目录，例如它可能是/etc/init.d/或者/etc/rc.d/rc.d。同时，对应每一个运行级别（runlevel）又有一个另外的目录，例如它们可能是/etc/rc2.d 或者 /etc/rc.d/rc2.d，这些目录中的文件通常是指向实际脚本文件的符号链接。

（2）直接在/etc/rc.d/rc.local 脚本中添加命令，该脚本应该在启动过程中调用。当然/etc/下的配置文件可能有不同的地方。

使用该种方法来配置应用程序自动启动，可以编辑“/etc/rc.d/rc.local”文件，将要执行

的程序（命令）添加到文件中。Linux 系统在启动后还未登录前，将自动执行该程序（命令），这相当于 Windows 下面的 autoexec.bat 文件。下面是一个在开机自动启动 mysql 数据库进程的/etc/rc.d/rc.local 脚本文件的内容。

```
#!/bin/sh
#
# This script will be executed *after* all the other init scripts.
# You can put your own initialization stuff in here if you don't
# want to do the full Sys V style init stuff.

touch /var/lock/subsys/local
/usr/local/mysql/bin/mysqld_safe --user=root &
```

举个例子来说，Linux 系统默认情况下在系统启动时不会自动开启 NFS（Network File System）服务，为了在系统每次重新启动时自动开启 NFS 服务，可以在"/etc/rc.d/rc.local"文件中加入下面的语句。

```
/etc/rc.d/init.d/nfs  restart
```

这样，在系统启动之前会自动打开 NFS 服务。再比如，如想开机时候就可以运行 apache 服务，则可以加下面语句到脚本文件/etc/rc.d/rc.local。

```
/usr/local/apache/apachectl  start
```

（3）通过/linuxrc 脚本直接启动，通常是在内核命令行参数中指定 init=/program。

Linux 内核一旦开始执行，它将通过驱动程序来初始化所有硬件设备，这初始化的过程可以在启动时从我们的 PC 机显示器的输出看出来，每个驱动程序都打印一些有关它的信息。初始化完成后，通常首先调用的是 init（通过 loader 向核心传入 init=/program 可以定制首先运行的程序）。比如在桌面 Linux 系统中，init 进程会读取/etc/inittab 文件，来决定执行级别和哪些脚本和命令。嵌入式应用开发中，可以根据实际的情况决定是否使用标准的 init 执行方式，也许这个 init 是个静态程序，它能够完成我们的嵌入应用的特定任务，那完全不用考虑 inittab 了，在这里可以采取比较灵活的措施。

“黑色经典”系列之《嵌入式 Linux 系统开发技术详解——基于 ARM》

FARSIGHT

第 11 章 充分利用开源软件

本章目标

本章介绍了嵌入式 Linux 系统常用的开源软件，包括系统工具、图形库、网络和串口应用程序等。通过学习本章介绍的开源软件，可以熟悉 Linux 系统所需要的各种软件，学会使用开源软件实现嵌入式 Linux 系统功能。

- Busybox 使用
- Linux 图形系统
- Linux 的网络应用
- Linux 的串口通讯

11.1　开放源代码工程介绍

首先来了解一下开放源代码软件的主要含义。简单地讲，开放源代码软件就是在开放源代码许可证下发布的软件，以保障软件用户自由使用及接触源代码的权利。这同时也保障了用户自行修改、复制以及再分发的权利，它源自黑客对智慧成果共享、自由的追求。概括地说，所有公布软件源代码的程序都可以称为开放源代码软件。

“开源”一词来源于 1997 年春天在美国加州的 Palo Alto 召开的一个所谓“纯粹程序员”参与的研讨会。在此次研讨会上，与会者一致通过了一个新的术语：OpenSource（开源软件）。也就是在这一年，开放源码促进会（Open Source Initiative，www.opensource.org）正式成立，这给予了开放源码一个官方的、正式的定义。它指出，开放源码并不只是意味着对源码的存取访问，而且还要遵守许多原则。开放源代码软件被定义为其源码可以被公众使用的软件，并且此软件的使用、修改和分发也不受许可证的限制。开放源码软件通常是有版权的，它的许可证可能包含这样一些限制：刻意保护它的开放源码状态，著者身份的公告，或者开发的控制。“开放源码”正在被公众利益软件组织注册为认证标记，这也是创立正式的开放源码定义的一种手段。

以下简单地回顾一下开源工程的发展历史。

- 1968 年，Internet 的先驱，ARPANET 建立。虽然 ARPANET 的设计目的仅仅是使研究人员在合作一个项目时可以共享代码和信息，但正是这样开启了开源文化的伟大历史进程。
- 1969 年，贝尔实验室的 Ken Thompson 研究员设计了 Unix 的第一个版本，这是一个多用户，多任务的操作系统。在整个 70 年代，Unix 的代码都在免费自由地传播，这也使得它迅速成为了在大学和研究机构中非常流行的操作系统。
- 1971 年，作为开放源码的先驱，Richard Stallman 加入了麻省理工学院的一个专门研究免费软件的组织。作为 Emacs 文本编辑器软件的开发者，他后来建立了 GNU（GNU's Not Unix）项目；这最终导致了免费的 Linux 操作系统的诞生。这也使得多少年来，每当人们提到开源文化的时候，就会首先想到其标志性人物—Richard Stallman，目前的 Project GNU 的计划主持人，Richard Stallman 后来成立 Free Software Foundation 组织，全力投入 Project GNU 的工作，被认为是 FSF 的终身义工。
- 1997 年，开放源码促进会（Open Source Initiative，www.opensource.org）正式成立，它给予了开放源码一个官方的、正式的定义。它指出，开放源码并不只是意味着对源码的存取访问，而且还要遵守许多原则。

开放源代码软件是在开放源代码许可证下发布的软件，以保障软件用户自由使用及接触源代码的权利。这同时也保障了用户自行修改、复制以及再分发的权利。简而言之，所有公布软件源代码的程序都可以称为开放源代码软件。开放源代码有时不仅仅指开放源代码软件，它同时也是一种软件开放模式的名称。比如 Linux 操作系统就是其中的一个代表。

注意　开源≠免费

11.1.1　Linux 系统和开源软件

Linux 是一个诞生于网络、成长于网络并且成熟于网络的操作系统。提到 Linux，就不能不说到开源文化，因为 Linux 的诞生与开源文化有着密切的关系。事实上，Linux 诞生的背景之一就是 GNU 项目计划和自由软件基金会（FSF），Linux 只是操作系统的一个内核，GNU 为其提供了软件环境。可以说，没有开源文化就没有 Linux 的诞生，甚至可以认为 Linux 就是开源精神的化身。接下来我们再来看看 Linux 操作系统的诞生过程。

- 1981 年 IBM 公司推出享誉全球的微型计算机 IBM PC。在 1981～1991 年间，MS-DOS 操作系统一直是微型计算机上操作系统的主宰。此时计算机硬件价格虽然逐年下降，但软件价格仍然是居高不下。
- 当时的另一个计算机软件技术的阵营是 Unix 世界。但是 Unix 操作系统价格非常的昂贵。为了寻求高利率，Unix 销商将价格抬得极高，PC 小用户就根本不可能有接触的机会。客观上需要一种能够服务于绝大多数 PC 机用户的操作系统。
- 1987 年，开发者 Andrew Tanenbaum 发布了 Minix，这是一个为 PC，Mac，Amiga，以及 Atari ST 设计的 Unix 版本，在发布时带有完整的源代码，并有一本详细的书本描述它的设计实现原理。由于 AST 的书写得非常详细，并且叙述有条有理，几乎全世界的计算机爱好者都在看这本书以理解操作系统的工作原理。
- 1991 年初，GNU 计划已经开发出了许多工具软件。最受期盼的 GNU C 编译器已经出现，但还没有开发出免费的 GNU 操作系统。即使是 MINIX 也开始有了版权，需要购买才能得到源代码。而 GNU 的操作系统 HURD 一直在开发之中，但并不能在几年内完成。当时，一名芬兰赫尔辛基大学计算机科学系的二年级学生—Linus Benedict Torvalds 为了克服在 MINIX 开发的局限性，开始着手开发一种新的并且是免费的操作系统。
- 1991 年的 10 月 5 日，Linus 在 comp.os.minix 新闻组上发布消息，正式向外宣布 Linux 内核系统的诞生（Free minix-like kernel sources for 386-AT）。这段消息可以称为 Linux 的诞生宣言，并且一直广为流传。3 年后，Linux 正式接受 GPL 公约。今天，按照 Red Hat Software 的说法，全球有大约 700 万 Linux 用户。正因如此，10 月 5 日对 Linux 社区来说是一个特殊的日子，许多后来 Linux 的新版本发布时都选择了这个日子。所以后来的 RedHat 公司选择这个日子发布它的新系统也并不偶然。
- 1994 年 3 月 14 日，正式的 Linux 内核 1.0 版本发布，约有 17 万行代码。它按完全自由免费的协议发布，源码必须完全公开，之后很快 Linux 正式采用 GPL 协议，这差不多是一种正式的独立宣言。截止那时，它的用户基数已经发展得很大，而且 Linux 的核心开发队伍也建立起来了。在 Linux 包含的数以千计的文件中，有一个名为 Credits，其中记录了主要的 Linux 黑客的姓名和电子函件地址。这个列表中包含了 100 多个名字的文件，世界各地的都有。此外，Linux 中包含有一系列的十分浅显易懂的 FAQ、Howto 和通用的帮助文件，这也是 Linux 发展史上的一座里程碑。
- 1996 年 2 月 9 日，Linux 内核 2.0 版本发布，可支持多个处理器，约由 40 万行代码。Linux 全球用户数约在 350 万左右。
- 1999 年 1 月，Linux 内核 2.2 测试版发布，预示着人们长久期待的正式稳定内核 2.2

即将发布。

- 2001 年 1 月 4 日，Linux 内核 2.4 正式发布。

11.1.2　开源软件的特点

开源软件（open source software ，OSS）的起源与 20 世纪 70 年代的黑客文化有关，经过二十多年的发展已经成为了软件产业发展的一个大方向。要理解什么是开源软件，先来了解一下什么是自由软件。自由软件英文为 Free software，其中的 free 既有免费的意思，也有自由的意思。对于自由软件的解释，自由软件运动的精神领袖理查德·斯托曼（Richard Stallman）曾经对“自由软件”的概念作过如下的阐述。

“自由软件”实际上指的是一种自由，而不是价格。为了理解这个概念，你应该想想“自由言论”，而不是“免费啤酒”。“自由软件”是指用户运行、拷贝、研究、改进软件的自由。而开源软件开发人员将自己的“产品”通过“GPL”协议（一种共享协议）提交给开源社区，开源模式促进了知识共享和交互，通过集体的智慧，不断帮助软件进行修改和完善。了解了自由软件可以更好地理解开源软件的概念。事实上，开源软件和自由软件之间的区别并不是很大，理查德·斯托曼也曾经说过，开源软件的优势，通常被形容为有实际价值的，可靠的软件，而自由软件运动则更崇尚使用自由。

概括来讲，开源软件主要有以下一些特点。

- 用户可以自行修改软件代码。
- 免费修改、免费重新发布，就是参与的人员可以直接了解各种开发和应用的方式，并且可以参与讨论，参与开发。
- 避免了传统模式需要花费大量成本的弊端，也降低了用户的应用成本，开源软件所使用的软件开发和应用都充分体现了“开放”和“共享”的思想。

对于开源软件的安全问题，开放源码软件支持者认为，每个人都能访问源代码，那么软件的漏洞就能够在最短的时间内被发觉，并且能够以最快的速度进行修复；而对于商业软件而言，只有内部人士才能够访问这些源代码程序，因此发现问题和解决问题相对都比较困难。事实上，影响系统安全的因素有很多，仅仅通过开放或封闭源代码，都不能从根本上解决安全问题。

11.2　Busybox 使用

11.2.1　Busybox 工程介绍

Busybox 工程于 1996 年发起，本身就是一个很成功的开源软件，其目的在于帮助 Debian 发行套件来建立磁盘安装。从 1999 年开始，此项目由 uClibc 的维护者 Erik Andersen 接手维护，起初是 Lineo 开源成果的一部分。Busybox 集成了一百多个最常用 Linux 命令（比如 init、getty、ls、cp、rm 等）和工具的软件，甚至还集成了一个 http 服务器和一个 telnet 服务器，并且支持 Glibc 和 uClibc，用户可以非常方便地在 Busybox 中定制所需的应用程序。使用

Busybox 可以有效地减小 bin 程序的体积，动态链接的 Busybox 工具一般在几百 KB 左右，而相对独立的 bin 程序加在一起的体积在几 M 左右甚至更大，这使得 Busybox 在嵌入式开发过程中具有不言而喻的优势。同时，使用 Busybox 可以大大简化制作嵌入式系统根文件系统的过程，所以 Busybox 工具在嵌入式开发中得到了广泛的应用。

11.2.2 配置编译 Busybox

下载 Busybox。最新版本的 Busybox 可以在器官方网站 www.busybox.net/download 下载，此处以下载的 Busybox-1.00-pre10 为例来说明。

以下是安装编译的详细过程。

1．拷贝 Busybox 源码压缩文件到指定目录，并解压

```
# cp /busybox-1.00-pre10.tar.gz /home
# cd /home
# tar xvfz busybox-1.00-pre10.tar.gz
# cd busybox-1.00-pre10
```

2．对 Busybox 进行配置，运行 make menuconfig 命令

```
# make menuconfig
```

配置界面如图 11.1 所示。



图 11.1 Busybox 编译配置界面

以下是 Busybox 配置菜单的主要选项列表。

---------------- BusyBox Configuration ----------------

General Configuration -- à

Build Options ----- à

Installation Options ----- à

Archival Utilities ----- à

Coreutils ----- à

Console Utilities ----- à

Debian Utilities ----- à

Editors ----- à

Finding Utilities ----- à

Init Utilities ----- à

Login/Password Management Utilities ----- à

Miscellaneous Utilities ----- à

Linux Module Utilities ----- à

Networking Utilities ----- à

Process Utilities ----- à

Another Bourne-like Shell ----- à

System Logging Utilities ----- à

Linux System Utilities ----- à

Debugging Options ----- à

下面就几个基本的选项配置进行说明，其余的部分可以根据开发的实际需要来定制。

（1）General Configuration 选项配置

General Configuration------ à

[*]Show verbose applet usage messages

[*]Runtime SUID/SGID configuration via /etc/busybox.conf

选中以上的两项。其配置界面如图 11.2 所示。



图 11.2　General Configuration 选项配置界面

（2）Build Options 选项配置

Build Options------ à

[*]Build BusyBox as a static binary(no shared libs)

[*]Build with Large File Support(for accesing files>2GB)

[　]Do you want to build BusyBox with a Cross Compiler

第一个选项是一定要选择的，因为选择静态编译可以把 Busybox 编译成静态链接的可执行文件，运行时独立于其他函数库，否则必需要其他共享库才能运行，此外采用静态编译也可以大大减少磁盘使用空间。

注意　如果只在 PC 机上使用 Busybox，不使用交叉编译功能，可以选中[ ]Do you want to build BusyBox with a Cross Compiler 选项，否则一定要选上。

该过程的编译界面如图 11.3 所示。

（3）Installation Options 选项配置

[*]Don’t use /usr

(./_install)BusyBox installation prefix

注意　这一项在编译过程中推荐选上，因为不选的话 Busybox 将默认安装到原系统的/usr 目录下面，这将覆盖掉系统原有的命令。

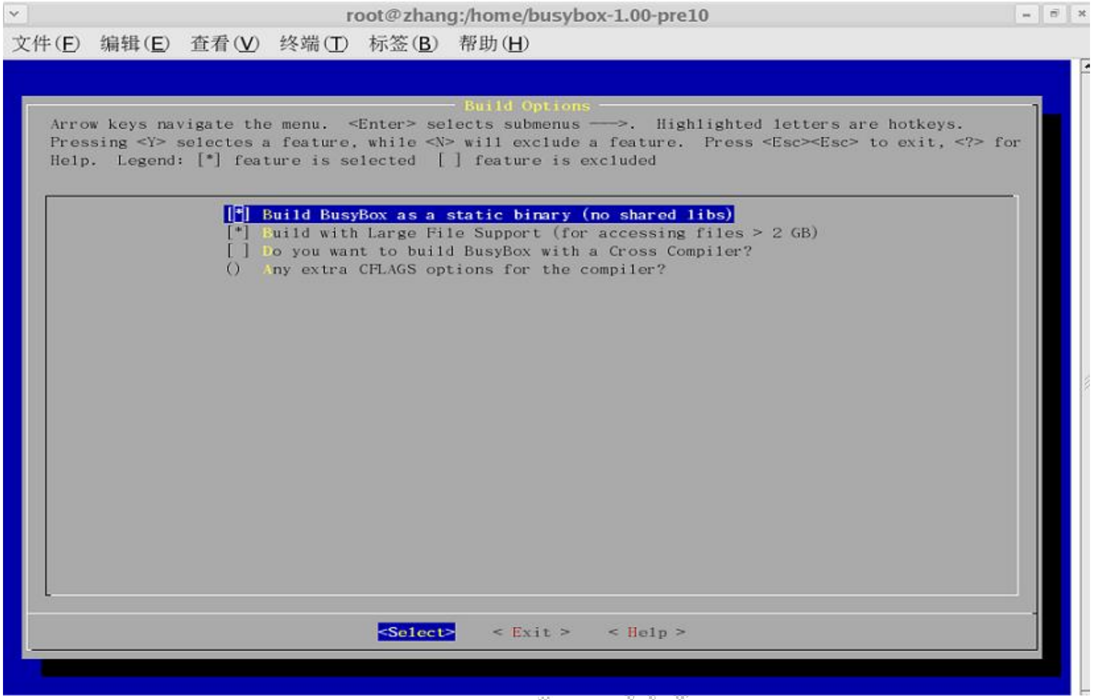


图 11.3 .Build Options 选项配置界面

该过程的编译界面如图 11.4 所示。



图 11.4 .Installation Options 选项配置界面

进行完基本的配置保存并退出之后配置信息保存在.config 配置文件，就可以继续下一步

的编译和安装工作了。

3．编译

运行编译命令即可，此处不再详述。

4．安装

如图 11.5 所示，make install 执行完之后会在 Busybox 安装目录下生成一个_install 的目录，里面有 Busybox 的可执行文件（/bin 中）和指向它的链接。

安装编译好 Busybox 之后，就可以使用了，生成的二进制可执行程序 Busybox 并不会直接被调用，而是通过指向它的符号链接来间接调用，可以有 2 种使用方法，下面举例来分别说明。

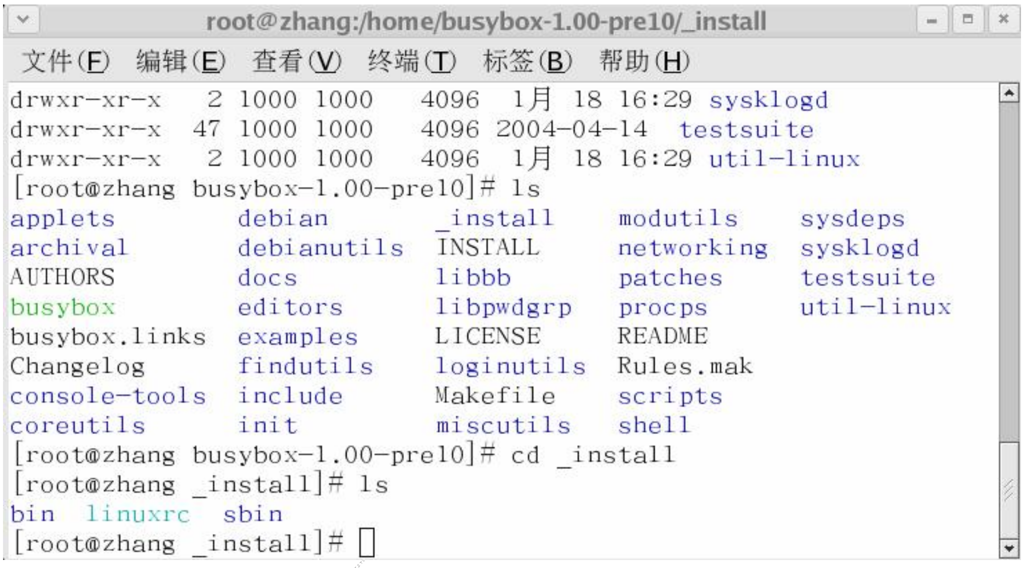


图 11.5　BusyBox 的_install 目录

（1）#……/bin/busybox　ls

其中，……/bin/busybox 是指可执行程序 Busybox 所在的目录，ls 是所要执行的程序，所以该用法就相当于常用的 ls 命令。

（2）Busybox 更加常用的用法是建立指向 Busybox 的符号链接，不同的链接名完成不同的功能，比如：

```
#ln -s……/busybox  ls
#ln -s…../busybox   rm
#ln -s…./busybox    cp
```

然后分别可以这样运行：

```
#  ./ls
#  ./rm
#  ./cp
```

这样就可以分别完成 ls、rm 和 cp 命令的功能。虽然都指向的是同一个可执行程序 Busybox，但是只要链接名不同，完成的功能就不同。对使用者来说，执行命令的方法并没有改变，命令行调用作为一个参数传给 Busybox，即可完成相应的功能。图 11.6 分别体现了

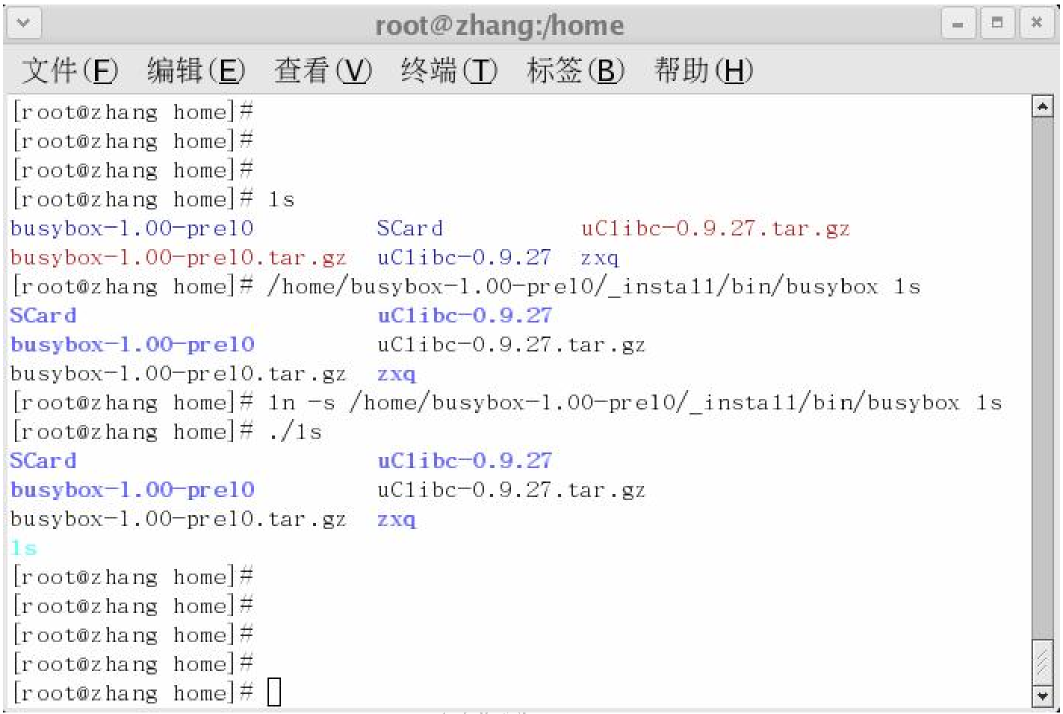


图 11.6 Busybox 使用方法

上述的 2 种用法。

11.3 X11 图形系统

11.3.1 X Windows 介绍

X Windows 系统于 1984 年在麻省理工学院（MIT）电脑科学研究室开始开发，当时 Bob Scheifler 正在发展分散式系统（distributed system）。同一时间，DEC 公司的 Jim Gettys 正在麻省理工学院做 Athena 计划的一部分。两个计划都需要一个相同的东西——一套在 Unix 机器上运行优良的视窗系统。因此合作关系开始展开，他们从斯坦福（Stanford）大学得到了一套叫作 W 的实验性视窗系统。因为是根据 W 视窗系统的基础开始发展的，当发展到了足以和原先系统有明显区别时，他们把这个新系统叫作 X 。

X Windows 系统也是一个基于客户/服务器（Client/Server）结构的窗口系统，在诞生之初是 Unix 系统上使用的图形用户界面，没有 PC 版，它允许在任一台 Unix 主机（客户端）上运行程序，而在基于 X 的终端（服务器）上显示出来。X Windows 系统是目前最常用的免费图形系统，配置在大多数的 Unix 系统、DEC 的 VAX/VMS 操作系统以及 Linux 系统中。

在后期又有了 XFree86 开发计划，其主要目的就是提供一个 PC 版的 X 窗口，主要移植到 Intel 的 x86 体系架构的处理器上，所以也称作是 XFree86 计划，它虽然不是以 GPL 授权，但是也可以自由拷贝以及传播，也可以使用在商业用途上。

X Windows 系统版本 11，也就是 X11 图形系统，产生于 1987 年。X11 图形系统是 X Windows 系统发展的一个重要里程碑。X11 是一个对网络透明的客户/服务器架构的图形显示系统，它支持应用程序在屏幕上绘制像素，线条，文字，图像等。X11 还包括一些其他的辅助的函数库，使得它可以容易地绘制用户界面，例如：按钮，文本输入区等。其组成主要有 3 部分：客户端；服务器和 X 协议。

严格地说，X Windows 系统并不是一个软件，而是一个协议（protocal）。这个协议定义一个系统成品所必需具备的功能（就如同 TCP/IP，DECnet 或 IBM 的 SNA，这些也都是协议，定义软件所应具备的功能）。任何系统能满足此协议及符合 X 协会其他的规范，便可称为 X 。

X11 是 Unix 的图形系统标准（X Window System）。Linux、各种 BSD 版本和多数的商用 Unix 都采用它。Linux 下的桌面图形系统已经发展得相当完善了，其 GUI 由窗口系统，窗口管理器，工具包和风格等几个部分组成，目前的桌面环境主要有 2 种。

1．KDE（K Desktop Environment）；

2．Gnome（GNU Network Object Model Environment）。

二者的界面非常相似。KDE 以 Qt 作为其底层库，而 Gnome 采用的是 GTK 库，Qt 最初并不遵从 GPL 协议，而 GTK 是完全遵守 GPL 宣言的，这也使得 Gnome 现在已经成为大多数 Linux 发行版本的首选，有关 GTK 和 Qt 会在后面的章节部分有详细的介绍。

11.3.2　Tiny-X 介绍

Tiny- X 是标准 X-windows 系统的简化版，去掉了许多对设备的检测过程，无需设置显示卡 Driver，很容易对各种不同硬件进行移植。Tiny- X 专为嵌入式开发，适合用作嵌入式 Linux 的 GUI 系统。Tiny- X 图形系统是由 SuSE 赞助的，开发人员是 XFree86 的核心成员 Keith Packard。目前 Tiny-X 是 XFree86 自带的编译模式之一，只要通过修改编译选项，就能编译生成 Tiny-X。

Tiny-X 作为 XFree86 4.0（ftp://ftp.xfree86.org/pub/XFree86/4.0）的子集，性能和稳定性都非常好，适合内存资源比较少的系统，它是以 XFree86 为基准，所以构置或设定的方式与 xfree86 是相同的。一般的 X Server 都太过于庞大，因此 Keith Packard 就以 XFree86 为基础，精简了不少东西而成 Tiny X Server，它的体积可以小到几百 KB 而已，非常适合应用于嵌入式环境。Tiny-X 像 X Window 系统一样采用标准的 Client/Server 体系结构，如图 11.7 所示。

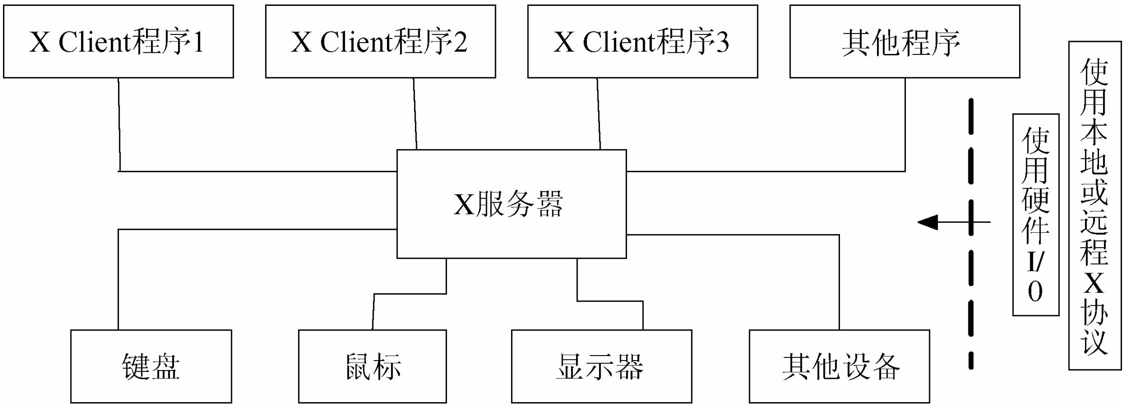


图 11.7 X-Window 的客户/服务器模型

（1）X 服务程序也称作显示管理器，是控制实际显示设备和输入设备的程序。它响应 X 客户程序的请求，直接与图形设备通信，负责打开和关闭窗口，控制字体和颜色等底层的具体操作。每一个显示设备只有一个惟一的 X 服务程序。

（2）X 客户程序是使用系统窗口功能的一些应用程序，无法直接影响窗口或显示，它们只能请求 X 服务程序，并通过 X 服务程序提供的服务在指定的窗口中完成特定的操作。

在嵌入式系统 GUI 开发中使用 Tiny-X 开发上层应用是比较方便的，在实际使用中，Tiny-X 底层要用到的库之间的关系如图 11.8 所示。

- Glib 类库：Glib 类库包括一些基本的数据类型和 C 语言需要的一些功能，与 GUI 无关，封装了一些常用的函数，如字符串相关函数、时间函数等。可以被 GDK 类库、GTK 类库或直接被应用程序调用。
- GDK 类库：建立在 Xlib 上的针对图形图像类封装的底层图形库。可以被 GTK 类库或应用程序直接调用。
- GTK 类库：建立在 Xlib 和 GDK 之上的面向对象的类库。GTK 提供了完善控件集，应用程序主要也是基于 GTK 类库来编写。

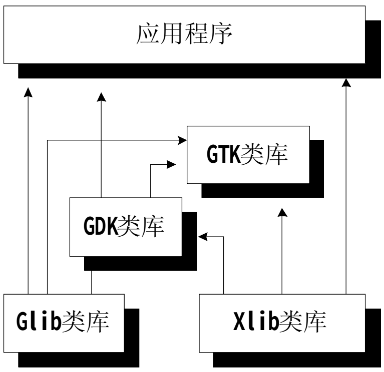


图 11.8 Tiny-X 库的调用关系

11.3.3 GTK 图形库

GTK（GIMP 工具箱，GIMP ToolKit）是一个功能强大而且快捷的开放源码图形库，用于 Unix/Linux 上的 X-Window 系统，程序员可以用来创建按钮、菜单及其他图形对象，GTK 最初是 GIMP 的专用开发库，后来发展为 Linux 下开发基于 X-Window 图形界面应用程序的主流开发工具之一，其本身就是自由软件，是 GNU 工程的一部分，因此可以用 GTK+开发开放源码软件、自由软件，甚至商业的、非自由的软件。Gtk 图形库使用一系列称为“构件（Widgets）”的对象来创建应用程序的图形用户接口。它提供了窗口、标签、命令按钮、开关按钮、检查按钮、无线按钮、框架、列表框、组合框、树、列表视图、笔记本、进度条等很多构件。使用 C 语言就可以用它们来构造丰富的用户界面程序。通

常情况下，用 GTK 代表软件包和共享库，用 GTK+代表 GTK 的图形构件集，现在的 GTK+中，相对以前的 GTK 来说包含了更多的标准回调机制来替代信号机制，符号‘+’就是用于区别原先的版本和新版本。在 Linux 下使用 GTK 开发 GUI 程序用 C 语言完成，发展到后来可以使用绑定了 C++语言的 GTKmm 工具来开发 GUI 程序，有关 GTKmm 的详细使用可参见相关书籍，此处不做详细的介绍。

GTK 是高层的库函数，它基本不使用 Xlib 库函数，而是使用更低层的函数库 GDK 和 Glib。这种结构使得 GTK 可以更方便地移植到其他系统上，或使用与 X-Windows 系统无关的图形库。图 11.9 是 GTK 库函数的结构图。

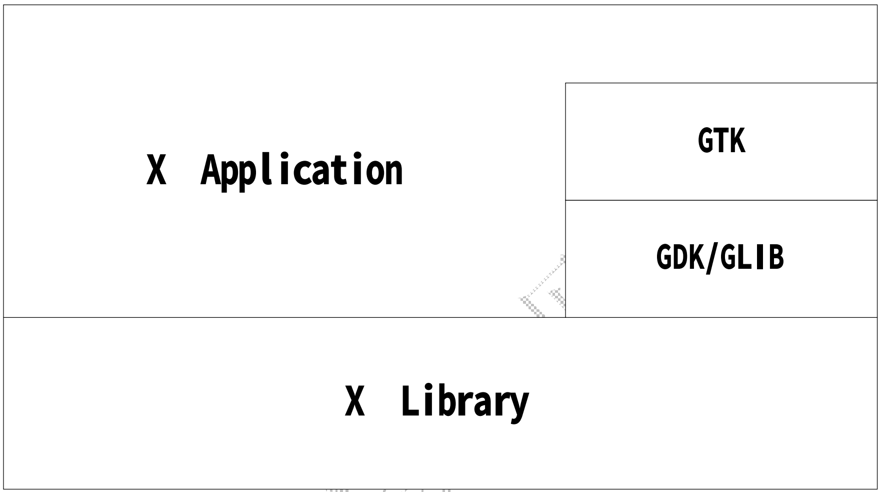


图 11.9　GTK 库函数结构

Glib 是一些与界面无关的函数构成的基本库，它定义了诸如基本类型、内存操作、定时器字符串操作等系列的函数以及一些宏定义，gdk 是底层的图形函数库，它包含 GTK 所使用的基本图形操作函数，比如基本图元、颜色、事件处理、图像和位图、窗口、拖放函数等。

GTK 具有以下的一些特性。

- 动态类型系统。
- 用 C 语言编写的对象系统，可实现继承、类型检验，以及信号/回调函数的基础结构。
- 类型和对象系统不是特别针对 GUI 的。
- GTKWidget 对象使用对象系统，它定义了 GTK+的图形组件的使用接口。
- 大量的 GTK Widget 子类（构件）。

要想用 GTK 编程，首先要保证系统中已经安装了 GTK 和 Gnome 库。编译安装 GTK 的过程很简单，如下所示。

（1）下载（ftp://ftp.gtk.org）GTK 安装包文件到指定目录，并解压，生成源码目录。

（2）进入源码目录，执行如下的操作。

```
./configure
```

该命令会生成编译时所需的 makefile 文件。

```
make
```

使用 make 命令来建立库。

```
make install
```

使用 make install 命令来安装库。

Gnome 的最新版本可以从 http://www.gnome.org 下载。取得新版本软件后，解压缩和安装的方法与 GTK 类似。安装好 GTK 及相关组件之后，就可以使用 GTK 来设计用户界面程序了。一般来讲，编写 GTK 程序主要有以下几个过程。

① 初始化
② 创建主窗口
③ 创建并加入子窗口
④ 设置组件回调
⑤ 显示窗口
⑥ 进入事件循环

下面举一个简单的例子来说明，创建一窗口标题为 hello 的窗体。

使用 Vi 创建一个 C 程序 hello.c，其代码如下。

```
#include <gtk/gtk.h>
int main( int argc,  char *argv[] )
{
     GtkWidget *window;         /****构件声明***/
     gtk_init (&argc, &argv);   /****初始化***/
     window = gtk_window_new (GTK_WINDOW_TOPLEVEL);  /****创建一个窗口***/
     gtk_widget_show  (window);              /****显示该窗口***/
     gtk_main ();
     return 0;
}
```

其中语句：

```
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_widget_show (window);
```

用来建立顶级窗口并且显示该窗口，它在缺省情况下是 200×200 像素大小。最后 gtk_main()使程序进入事件循环阶段，GTK 将在内部处理事件。

接下来就可以编译该 GTK 程序了，使用下面的命令。

```
#[root@localhost] gcc -Wall –g  hello.c -o  hello  'gtk-config --cflags' 'gtk-config --libs'
```

注意 编译命令中使用的引号是后置引用（backquote），反引号中的字符串实际就是一个命令，你的 Shell 会执行它。

在编译程序的命令中，'gtk-config --cflags'产生编译 GTK 所使用的头文件位置，'gtk-config --libs'产生连接 GTK 程序所使用的库。一般地，涉及 GTK 的库有：libgtk.so libgdk.so libgmodule.so libglib.so。

编译成功之后，就可以执行了，执行之后就会看到一个简单的窗体，如图 11.10 所示。

```
./hello
```

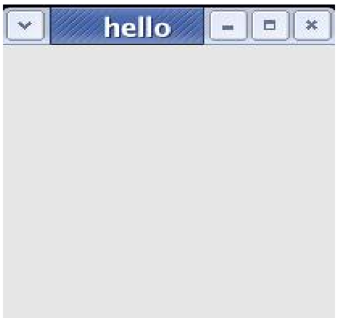


图 11.10　GTK 编程示例

Linux 在 GTK 发展到一定阶段时，还没有像 Windows 平台上的 Visual Basic、Delphi 等一样的可视化的应用程序开发工具。开发 Linux 的 GUI 应用程序需要用文本编辑器书写源代码，然后再用编译器生成应用程序，这样做往往给开发人员带来繁重的工作，特别是在使用不同的布局构件组装界面元素，创建菜单、工具条等时，不能在编写代码时直接看到显示效果；其次是对代码量较大的程序，可能要将代码放在不同的 C 语言文件中；为它们配置编译选项、编写 Makefile 文件也是一项巨大的工程，特别是对于大型项目的开发。

Glade 就在这样的背景下产生了，Glade 是一种 GUI 生成器，它是传统界面设计工具 GTK/GDK 的扩展，可以快速生成创建界面的 C 源程序。Glade 用来为 GTK+和 GNOME 程序快速地设计图形用户界面。如果安装了 GTK+和/或 GNOME 库，也可用在 Linux 下的任何桌面环境中。

Glad 可以用非常直观的界面和可视化的方法来设计应用程序界面，设置窗口、构件的外观、设置构件信号的回调函数，然后生成 C 语言代码（事件响应代码需要手工添加），这无疑使得在 Linux 下开发 GUI 软件更加方便和快捷，下面简单介绍一下使用 Glade 界面生成器来设计图形用户界面的流程。

1．Glade 界面介绍

启动 Glade 界面生成器之后，会看到以下 3 个窗体。

（1）主界面窗体，如图 11.11 所示

主窗口显示应用程序的最顶层对象，如窗口、弹出菜单、对话框等。要编辑这些对象时，只需在主窗口的列表中双击该对象，即可打开它。选中对象，按 Delete 键，就可以将该对象删除。其菜单选项有：“新建”、“打开”、“保存”、“编辑”、“查看”等。

（2）构件箱窗口，如图 11.12 所示

可以从图 11.12 中看到 Glade 带有丰富的构件，构件箱中容纳了绝大多数 GTK+/Gnome 构件。点击构件箱上的构件，再在窗口上点击，可以将构件添加到窗口上。要注意的是，在窗口上添加构件要首先添加容器再放置相应控件；构件箱将 GTK+/Gnome 构件分为 3 类：GTK+基本构件、GTK+附加构件以及 Gnome 构件。在构件箱上点击任何一个标签页，都将显示上面 3 类构件中的构件。选择某个构件后，点击 Selector 前的箭头，可以取消前面的选择。

（3）属性编辑器窗口，如图 11.13 所示



图 11.11 Glade 主界面窗体

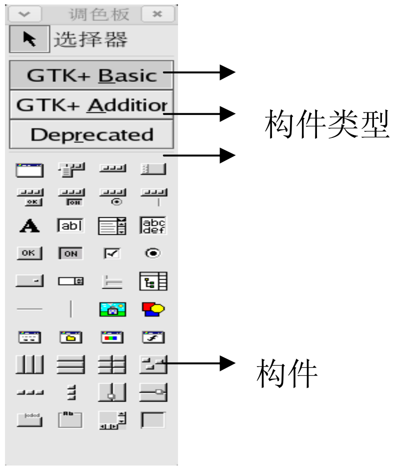


图 11.12 Glade 构件箱窗体

在主窗口的“查看”菜单中选择“显示属性编辑器”（默认情况下选中），可以打开“属性编辑器”窗口。属性编辑器用于设置构件的属性。比如窗口的缺省尺寸、窗口的标题等。这里可以设置构件的名称、构件在窗口上的组装位置、构件的加速键，还可以为构件的信号设置回调函数。

2．使用 Glade 生成图形用户界面

下面以一个简单的例子来介绍使用 Glade 生成图形用户界面的过程。

最终生成下面这样的一个界面，包含一个名称为 MyWindow 的窗体和一个按钮。如图 11.14 所示。



图 11.13 Glade 属性编辑窗口



图 11.14 一个简单的 Glade 设计

当然，上面的设计也完全可以通过前面提到的方法，即使用 GTK 语言来完成，现在主

要通过使用 Glade 生成器来自动实现源代码。

（1）单击 Glade 主界面窗体的“新建”按钮来建立一个新的 project，并且保存到指定目录。

（2）创建一个新的窗体

在构件箱窗体上的 Gtk+ Basic 类型下单击 GtkWindow 构件，出现一个窗口标题为 window1 的窗体，然后在属性编辑器中将其标题改为“MyWindow”。

（3）在窗口中添加一个构件——按钮，在属性编辑器里设置其属性，比如名称为“确定”。

（4）为构件设置信号回调函数。

点击属性编辑器的“信号”选项，添加一个信号回调函数，比如，在 signal 一栏中选择一个 clicked 信号（不同的构件有不同的 signal），在 Handler 一栏中选择一个函数，如：0n_button1_clicked。

（5）生成源代码

完成以上的工作就可以编译该工程文件（以.glade 后缀）来生成源代码了，具体方法是选择 Glade 主界面窗体的 Project 选项中的“build”选项（Ctrl+B），之后就会在指定保存的目录下生成整个工程文件（包括源文件），这些源代码是进行深层次开发的基础。通常情况下，编译之后会生成一个 src 目录来存放源代码文件，如果想在编译之前知道 build 后的源文件都有哪些，可以在 Glade 主界面窗体中选择“选项”中的“C 选项”，如图 11.15 所示。



图 11.15　Glade 项目源文件组成

在图 11.15 中可以看到，编译之后会生成 3 种类型的源文件。

- 界面创建函数（名称可以修改）

Interface.c（源文件）：创建用户界面的函数。

Interface.h（头文件）：interfac.c 中的所有函数声明。

- 信号处理函数和回调函数

Callbacks.c（源文件）：所有的回调函数。

Callbacks.h（头文件）：函数声明。

- 支持函数

Support.c（源文件）：包含 Glade 的实用函数。

Support.h（头文件）：函数声明。

源代码生成以后，还需要编译才能生成最终的可执行程序，可以查看 Glade 项目文件的组成都有哪些，进入相应目录可以查看，如图 11.16 所示。

从图 11.16 可以看出，Glade 项目文件的组成比较多，即便通过创建 Makefile 文件来编译也是很复杂的，一般使用 GNU 工具来自动创建 Makefile 文件。Glade 所生成的源代码中有一个名为 autogen.sh 的脚本文件（如图 11.15 所示）来完成这一工作，即自动生成该项目的 Makefile 文件，比如可以在 Glade 项目文件所在的目录下执行如下的指令：

```
# ./autogen.sh
```

执行该指令后，autogen.sh 脚本会自动搜索源文件的路径、头文件及依赖库的路径，然后生成一个 Makefile 文件，这时就可以开始编译源代码了。

```
# make
```

编译完之后，就可以在 Glade 项目文件的 src 目录下看到最终的可执行程序 mywin 了。在该目录下执行 mywin 就可以看到图 11.14 的效果了。

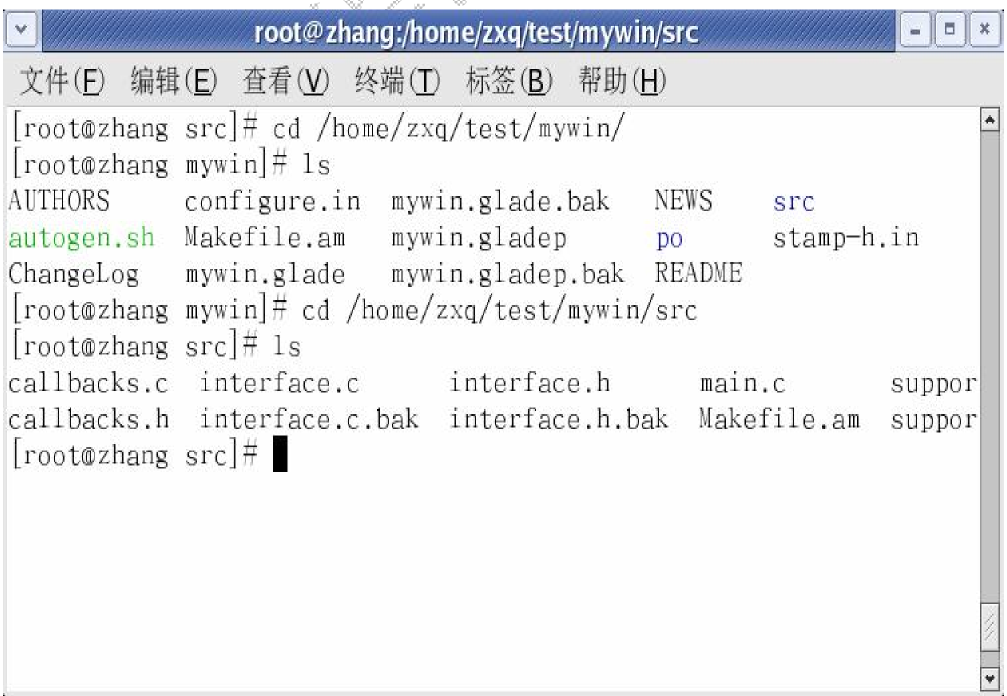


图 11.16　Glade 项目文件组成

11.4　Qt 图形库

11.4.1　Qt 介绍

Qt 是由挪威 TrollTech 公司开发的跨平台 C++图形用户界面开发工具，也是该公司的一个标志性产品，有商业版和免费的版本两种。程序开发员利用 Qt 可以编写单一代码的应用程序，并可在 Windows、Linux、Unix 及 Mac OS X 和嵌入式 Linux 等不同平台上进行本地化运行。目前，Qt 已被成功地应用于全球数以千计的商业应用程序。此外，Qt 还是开放源代码 KDE 桌面环境的基础。TrollTech 公司在 1995 年推出了 Qt 的第一个商业版本，直到现在 Qt 已经被世界各地跨平台的开发人员所使用，而 Qt 的功能也得到了不断的完善和提高。Qt 以工具开发包的形式提供给开发者，这些开发包包括了图形设计器、Makefile 制作工具、字体化国际工具和 Qt 的 C++类库等，Qt 的一个显著特点是跨平台特性，目前 Qt 支持的操作系统平台包括以下几种。

- Windows 系列
- Unix/Linux/Solaris
- 包含有 FramBuffer（帧缓冲）的嵌入式 Linux 平台
- Macintoshi Mac OSX

Qt 对不同平台（Unix，Windows，Mac）的专门 API 进行了封装，如文件处理、网络（操作，协议），进程处理、线程、数据库访问等。Qt 的类库也等价于 Windows 平台下的 MFC 开发库，但是 Qt 的类库是支持跨平台的类库，它封装了可以适应不同操作系统的访问细节。

从本质上讲，Qt 同 X Window 上的 Openwin，GTK 等图形界面库和 Windows 平台上的 MFC，OWL，VCL，ATL 是同类型的东西，但是 Qt 也具有其独特的优点。

- 优良的跨平台特性
- 面向对象特性

Qt 的良好封装机制使得 Qt 的模块化程度非常高，可重用性较好，对于用户开发来说是非常方便的。Qt 提供了一种称为 signals/slots（信号与插槽机制）的安全类型来替代 callback，这使得各个元件之间的协同工作变得十分简单。

- 丰富的 API

Qt 包括多达 250 个以上的 C++类，还提供基于模板的 collections，serialization，file，I/O device，directory management，date/time 类。甚至还包括正则表达式的处理功能。

- 支持 2D/3D 图形渲染，支持 OpenGL
- 大量的开发文档
- XML 支持

如果系统中安装了 Qt 之后，可以看到其启动界面如图 11.17 所示。

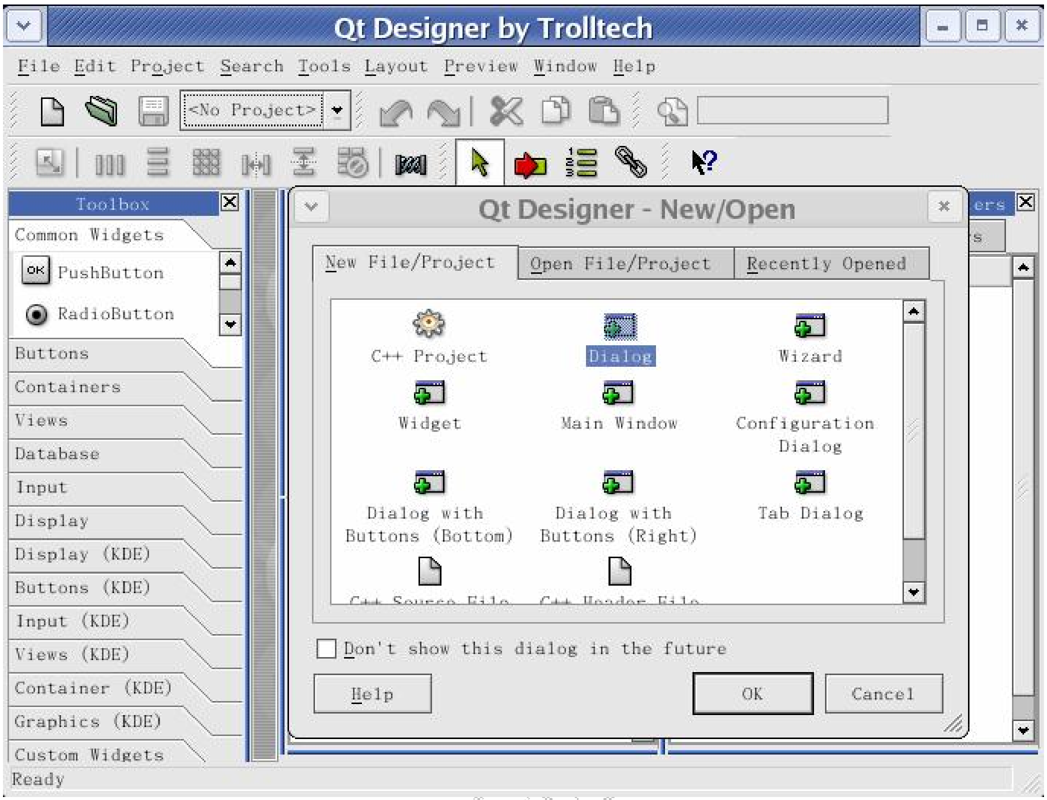


图 11.17 Qt 启动界面

11.4.2 Qt/Embedded 介绍

随着嵌入式 Linux 应用的不断发展，嵌入式处理器运算能力的不断增强，越来越多的嵌入式设备开始采用较为复杂的GUI系统，尤其是手持设备中的GUI系统发展得非常迅速，在这个过程中Qt/Embedded得到了广泛的应用。Qt/Embedded是著名的Qt库开发商Trolltech正在进行的、面向嵌入式系统的Qt版本，是一个专门为小型设备提供图形用户界面的应用框架和窗口系统，被看作是Qt版本的嵌入式Linux端口，也是完整的自包含C++GUI和基于Linux的嵌入式平台开发工具。它为开发者提供了丰富的窗口构件（Widgets），并且还支持窗口部件的定制，因此可以为用户提供非常丰富的图形界面。Qt 是 KDE 等项目使用的GUI 支持库，开发人员多为 KDE 项目的核心开发人员，所以有许多基于 Qt 的 X Window 程序可以很容易地移植到 Qt/Embedded 版本上，目前的应用也很广泛，包括各种消费电器和工业控制设备。

Qt/Embedded 是一个专为访问嵌入式设备的 API，Qt/Embedded 同样采用的是 Server/Client 结构，Qt/Embedded 类库完全采用 C++封装。丰富的控件资源和较好的可移植性是 Qt/Embedded 最为优秀的一方面，它的类库接口完全兼容于同版本的 Qt-X11，使用 X 下的开发工具可以直接开发基于 Qt/Embedded 的应用程序。作为针对嵌入式系统的一款优秀图形界面开发工具，以下的几个方面值得关注。

1．Qt/Embedded 采用 framebuffer 作为其底层图形引擎

Qt/Embedded 以原始 Qt 为基础，并做了许多出色的调整以适用于嵌入式环境。Qt/Embedded 通过 Qt 的 API 与 Linux I/O 设施直接交互，成为嵌入式 Linux 端口。同 Qt/X11 相比，Qt/Embedded 很省内存，因为它不需要一个 X 服务器或是 Xlib 库，它在底层采用 framebuffer（帧缓冲）作为底层图形引擎，framebuffer 是出现在 Linux2.2.x 以上内核的本当中的一种驱动程序接口。这种接口采用 mmap 系统调用，将显示设备抽象为帧缓冲区。用户可以将它看成是显示内存的一个映象，将其映射到进程地址空间之后，就可以直接进行读写操作了，而且写操作可以立即反映在屏幕上。framebuffer 驱动程序是最重要的驱动程序之一，正是这个驱动程序才使系统屏幕能显示内容。

2．Qt/Embedded 的事件驱动基础

在任何 GUI 系统中，均有事件或消息驱动的概念。Qt/Embedded 中与用户输入事件相关的信号，是建立在对底层输入设备的接口调用之上的。Qt/Embedded 中的输入设备，分为鼠标类与键盘类。Qt/Embedded 3.0 支持的几种鼠标协议有：BusMouse、IntelliMouse、Microsoft 和 Mouseman。其中鼠标设备的抽象基类为 QWSMouse Handler，从该类又重新派生出一些具体的鼠标类设备的实现类。Qt/Embedded 支持标准的 101 键盘和 Vr41XX 按键，通过抽象基类 QWSKeboardHandler 可以让 Qt/Embedded 支持更多的客户键盘和其他的非指示设备。

目前，Qt/Embedded 采用 2 种方式进行发布：在 GPL 协议下发布的 free 版本与专门针对商业应用的 commercial 版本。二者除了发布方式不同之外，在源码方面没有任何区别。随着嵌入式处理器运算能力的不断提高，对外设支持的不断丰富，嵌入式 Linux 系统下的 GUI 开发也越来越深入。由于 Qt/Embedded 延续了 Qt 在桌面系统的所有功能，丰富的 API 接口和基于组件的编程模型使得嵌入式 Linux 系统中的应用程序开发更加方便，再加上 Qt/Embedded 本身就面向高端的手持设备和移动设备，这使得 Qt/Embedded 在未来有更广阔的发展前景。

11.4.3　Qt/Embedded 架构

1．窗口系统

一个 Qt/Embedded 窗口系统包含了一个或多个进程，其中的一个进程可作为服务器。这个服务进程会分配客户显示区域，以及产生鼠标和键盘事件。这个服务进程还能为已经运行的客户程序提供输入方法和用户接口。这个服务进程其实就是一个有某些额外权限的客户进程。任何程序都可以在命令行上加上“-qws”的选项来把它作为一个服务器运行。

客户与服务器之间的通信使用共享内存的方法实现，通信量应该保持最小，例如客户进程直接访问帧缓冲来完成全部的绘制操作，而不会通过服务器，客户程序需要负责绘制它们自己的标题栏和其他式样。这就是 Qt/Embedded 库内部层次分明的处理过程。

客户可以使用 QCOP 通道交换消息。服务进程简单的广播 QCOP 消息给所有监听指定通

道的应用进程，接着应用进程可以把一个插槽连接到一个负责接收的信号上，从而对消息做出响应。消息的传递通常伴随着二进制数据的传输，这是通过一个 QDataStream 类的序列化过程来实现的。

2．字体

Qt/Embedded 支持 4 种不同的字体格式：True Type（TTF）、Postscript Type1、位图发布字体（BDF）和 Qt 的预呈现（Pre-rendered）字体（QPF）。Qt 还可以通过增加 QFontFactory 的子类来支持其他字体，也可以支持以插件方式出现的反别名字体。

每个 TTF 或者 TYPE1 类型的字体首次在图形或者文本方式的环境下被使用时，这些字体的字形都会以指定的大小被预先呈现出来，呈现的结果会被缓冲。根据给定的字体尺寸（例如 10 或 12 点阵）预先呈现 TTF 或者 TYPE1 类型的字体文件并把结果以 QPF 的格式保存，这样将可以节省内存和 CPU 处理时间。QPF 文件包含了一些必要的字体，这些字体可以通过 makeqpf 工具取得，或者通过运行程序时加上“savefonts”选项获取。如果应用程序中使用到的字体都是 QPF 格式，那么 Qt/Embedded 将被重新配置，并排除对 TTF 和 TYPE1 类型的字体的编译，这样就可以减少 Qt/Embedded 的库的大小和存储字体的空间。例如一个 10 点阵大小的包含所有 ASII 字符的 QPF 字体文件的大小为 1300 字节，这个文件可以直接从物理存储格式映射成为内存存储格式。

Qt/Embedded 的字体通常包括 Unicode 字体的一部分子集，ASII 和 Latin-1。一个完整的 16 点阵的 Unicode 字体的存储空间通常超过 1M，我们应尽可能存储一个字体的子集，而不是存储所有的字。

3．输入设备

Qt/Embedded 3.0 支持以下几种鼠标协议。

- BusMouse
- IntelliMouse
- Microsoft
- MouseMan

通过从 QWSMouseHandler 或 QcalibratedMouseHandler 派生子类，Qt/Embedded 可以支持更多的客户指示设备；通过子类化 QWSKeyboardHandler，Qt/Embedded 可以支持更多的客户键盘和其他的非指示设备。

4．输入方法

需要注意的是：对于非拉丁语系字符（例如阿拉伯文、中文、希伯来文和日文）的输入法，需要把它写成过滤器的方式，并改变键盘的输入。

5．信号与插槽机制

信号与插槽机制提供了对象间的通信机制，它易于理解和使用，并完全被 Qt 图形设计器所支持。图形用户接口的应用需要对用户的动作做出响应。例如，当用户点击了一个菜单

项或是工具栏的按钮时，应用程序会执行某些代码。大部分情况下，我们希望不同类型的对象之间能够进行通信。程序员必须把事件和相关代码联系起来，这样才能对事件做出响应。以前的工具开发包使用的事件响应机制是易崩溃的，不够健壮的，同时也不是面向对象的。Trolltech 已经创立了一种新的机制，叫作“信号与插槽”。它是一种强有力的对象间通信机制，它完全可以取代原始的回调和消息映射机制；信号与插槽是迅速、类型安全、健壮、完全面向对象并用 C++实现的一种机制。

如果在其他设计中采用回调函数（callback）机制关联某段响应代码和一个按钮的动作时，通常需要把该段响应代码写成一个函数，然后把这个函数的地址指针传给按钮，当那个按钮被按下时，这个函数就会被执行。对于这种方式，以前的开发包不能够确保回调函数被执行时所传递进来的函数参数就是正确的类型，因此容易造成进程崩溃，另外一个问题是，回调这种方式紧紧的绑定了图形用户接口的功能元素，因而很难把开发进行独立的分类。

Qt 的信号与插槽机制是不同的。Qt 的窗口在事件发生后会激发信号。例如一个按钮被点击时会激发一个“clicked”信号。程序员通过建立一个函数（称作一个插槽）然后调用 connect()函数把这个插槽和一个信号连接起来，这样就完成了一个事件和响应代码的连接。信号与插槽机制并不要求类之间互相知道细节，这样就可以相对容易的开发出代码可高重用的类。信号与插槽机制是类型安全的，它以警告的方式报告类型错误，而不会使系统产生崩溃。

11.4.4 Qt/Embedded 软件包与安装

在 Linux 下安装 Qt/Embedded 开发环境主要需要 3 个软件：tmake 工具安装包；Qt/Embedded 安装包和 Qt 的 X11 版的软件安装包。以下列举了上述几个软件包及其版本号。

- Tmake 1.11（或更高版本）软件包（主要用来生成 Qt/Embedded 应用工程的 Makefile 文件）
- Qt/Embedded 2.3.7（Qt/Embedded 的安装包）
- Qt 2.3.2 for X11（Qt 的 X11 版的安装包，它将产生 X11 开发环境所需的 2 个工具）

以上这些软件可以从 trolltech 的 Web 或 FTP 服务器上免费下载。

注意 由于上述软件安装包有许多不同的版本，而由于版本的不同可能会造成潜在的冲突而影响使用，一个基本的原则是：选择的 Qt for X11 的安装包应该比 Qt/Embeedded 的安装版本要旧，这是由于 Qt for X11 安装包的 2 个工具 uic 和 designer 产生的的源文件会和 Qt/Embedded 的库一起被编译链接，也就是考虑到“向前兼容”的原则。

目前，Qt/Embedded 可以运行在 Linux 所支持的各种处理器上，包括像 Intel X86、ARM、MIPS 和 PowerPC 等处理器上。Qt/Embedded 对内存的消耗很低，因为它不需要 X 服务器或是 Xlib 库，可以直接写缓冲帧，对于不使用的功能可以在编译的时候动态调节从而尽可能的减少对内存的使用。例如，在实际使用过程中不想使用 QlistView 这个库时，可以通过定义一个 QT_NO_LISTVIEW 的预处理标记来实现。它甚至可以把全部的应用功能编译链接到一个简单的静态链接的可执行程序中。Qt/Embedded 提供了大约 200 多个可配置的特征，在 Intel

X86 平台上库的大小范围会在 700～5000KB 之间。

以下是上面几个软件包的安装过程。

1．安装 tmake

在终端下运行以下命令。

```
tar xfz tmake-1.11.tar.gz
export TMAKEDIR=$PWD/tmake-1.11
export TMAKEPATH=$TMAKEDIR/lib/qws/linux-x86-g++
export PATH=$TMAKEDIR/bin:$PATH
```

2．安装 Qt/Embedded 2.3.7

在终端下运行以下命令。

```
tar xfz qt-embedded-2.3.7.tar.gz
cd qt-2.3.7
export QTDIR=$PWD
export QTEDIR=$QTDIR
export PATH=$QTDIR/bin:$PATH
export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
./configure -qconfig -qvfb -depths 4,8,16,32
make sub-src
cd ..
```

上述命令./configure -qconfig -qvfb -depths 4，8，16，32 指定 Qt 嵌入式开发包生成虚拟缓冲帧工具 qvfb，并支持 4，8，16，32 位的显示颜色深度。另外也可以在 configure 的参数中添加 system，jpeg 和 gif，使 Qt/Embedded 平台能支持 jpeg、gif 格式的图形。

命令 make sub-src 指定按精简方式编译开发包，也就是说有些 Qt 类未被编译。Qt 嵌入式开发包有 5 种编译范围的选项，使用这些选项，可控制 Qt 生成的库文件的大小，但是您的应用所使用到的一些 Qt 类将可能因此在 Qt 的库中找不到链接。编译选项的具体用法可运行./configure－help 命令查看。

3．安装 Qt/X11 2.3.2

在终端下运行以下命令。

```
tar xfz qt-x11-2.3.2.tar.gz
cd qt-2.3.2
export QTDIR=$PWD
export PATH=$QTDIR/bin:$PATH
export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
./configure -no-opengl
```

```
Make
make -C tools/qvfb
mv tools/qvfb/qvfb bin
cp bin/uic $QTEDIR/bin
cd ..
```

11.5 MiniGUI 图形系统

11.5.1 MiniGUI 图形系统概述

MiniGUI 是一个面向实时嵌入式系统或者实时系统的轻量级图形用户界面支持系统，是我国为数不多的在国际比较知名自由软件之一。MiniGUI 遵循 GPL 条款发布，其目标是为嵌入式 Linux 系统建立一个快速、稳定和轻量级的用户界面支持系统。经过几年的发展，MiniGui 已经成为一个非常成熟和稳定的图形系统，并且在许多实际产品或项目中得到了广泛应用，诸如手持信息终端、机顶盒、工业控制系统及工业仪表、金融终端等产品和领域。

MiniGui 稳定版本的主要特征如下。

- 是遵循 GPL 条款的自由软件；
- 提供完备的多窗口机制和消息传递机制；
- 提供丰富的控件，如静态文本框、按钮、列表框、组合框、进度条、属性页、工具栏、拖动条树形控件等；
- 对话框和消息框；
- 其他 GUI 元素，包括菜单、加速键、插入符、定时器等；
- 界面皮肤支持；
- Windows 资源文件支持，如位图、图标、光标等；
- 各种流行图像文件的支持，包括 JPEG、GIF、PNG、BMP 等；
- 多字符集和多字体支持，目前支持 ISO-1～ISO8859-15、GB2312、GBK、GB18030 等字符集；
- 多种键盘布局支持；
- 支持汉字（GB2132）输入法支持，包括内码、全拼、智能拼音；
- 增强的新的 GDI 函数，包括光栅操作、复杂区域处理、椭圆、圆弧、多边形及区域填充函数。

MiniGUI 本身的占用空间非常小，以嵌入式 Linux 操作系统为例，一个典型的 MiniGUI 系统存储空间占用情况如表 11.1 所示。

表 11.1　　典型 MiniGui 系统空间占用情况

组 成 部 分	容　　量	说　　明
Linux 内核	300K～1M	由系统决定

MniGUI 支持库	500K～700K	由编译选项确定
MniGUI 字体、位图等资源	400K	由应用程序确定，可缩小到 200K 以内
GB2312 输入法码表	200K	不是必需的，由应用程序确定
应用程序	1M～2M	由应用程序决定

从上表可以看到，系统总体的占有空间应该在 2MB～5MB 左右。在某些系统上，功能完备的 MiniGUI 系统本身所占用的空间可进一步缩小到 1MB 以内，所以比较适合嵌入式系统。

MiniGUI 具有良好的软件架构，通过抽象层将 MiniGUI 上层和底层操作系统隔离开来如图 11.18 所示。

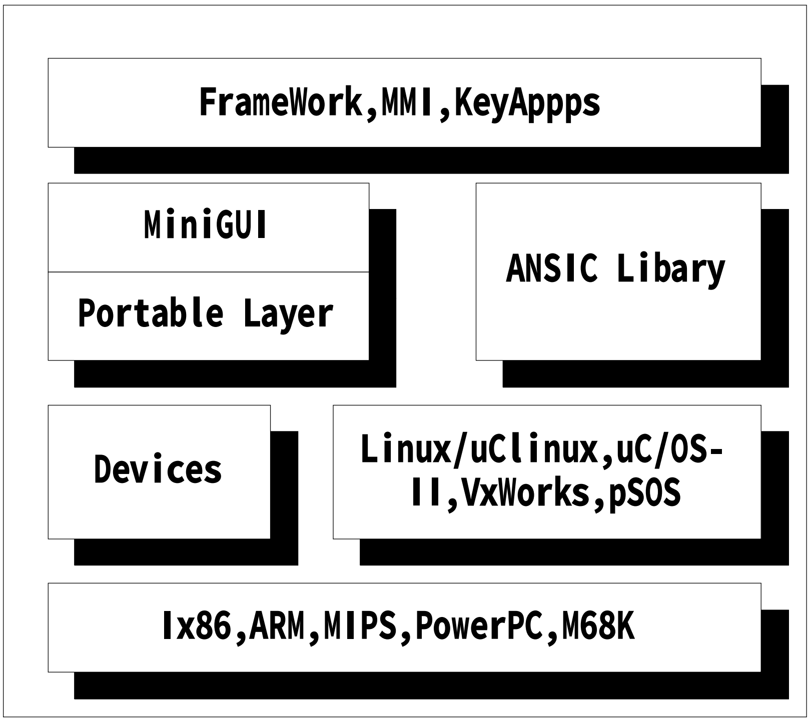


图 11.18　MiniGUI 层次结构图

如图 11.18 所示，基于 MiniGUI 的应用程序一般通过 ANSI C 库以及 MiniGUI 自身提供的 API 实现其自身的功能；MiniGUI 中的“Portable Layer（可移植层）”可将特定操作系统及底层硬件的细节隐藏起来，而上层应用程序则无需关系底层的硬件平台输出和输入设备，支持的硬件平台包括 Intel x86、ARM、PowerPC、MIPS、M68K 等，并且 MiniGUI 已成为跨操作系统的图形用户界面支持系统，目前可在 Linux/uClinux、eCos、uC/OS-II、VxWorks 等操作系统上运行。总的来说，MiniGui 是一个非常适合于工业实时控制以及嵌入式系统的高效、可靠、可定制、小巧的图形用户支持系统。在今后的发展中会得到更加广泛的应用。

11.5.2 MiniGUI 移植

下面主要介绍把 MiniGUI 移植到和本教材配套的优龙 FS2410 开发板上。运行 MiniGUI 系统需要满足一些前提条件。

- 支持 POSIX.X 的 Linux 系统，包括 Linux2.0、Linux2.4，也包括 uClinux 等非标准的 Linux 系统；
- Linux 的 Framebuffer 驱动程序功能正常添加，对于没有 Framebuffer 支持的 Linux 系统，需要编写特定的图形引擎才能运行 MiniGUI；
- 运行 MiniGUI-Threads 版本需要 POSIX 兼容线程库的支持；
- 运行 MiniGUI-Lite 版本需要 UNIX Domain 套接字机制的支持。

下面是编译的整个过程。

1．搭建开发环境

```
mkdir -P /home/uc2410/target/
export $PREFIX=/home/uc2410/target/
cd $PREFIX
mkdir /mnt/tmp
mount -t jffs2 rootfs-0.9.26.jffs2 /mnt/tmp -o loop
mount -t cramfs root_china.cramfs/mnt/tmp -o loop
cp -aR /mnt/tmp ./root_china
```

2．编译 MiniGUI

编译所采用的软件是：libminigui1.3.3，mde-1.3.0 和 minigui-res-1.3.3。libminigui 目录下提供了很多交叉编译脚本，参考并编辑一个适合我们的脚本 build-2410，其程序清单如下。

```
#!/bin/bash
rm config.cache config.status -f
CC=arm-linux-gcc \
./configure --prefix=$PREFIX \
--build=i386-linux \
--host=$TARGET \
--target=$TARGET \
--disable-debug \
--disable-static \
--enable-lite  \
--disable-galqvfb \
--enable-newgal \
--enable-purefbgfx \
```

```
--disable-nativegal \
$rot_dir \
$ial_dir \
--disable-nativeial \
--disable-pcxsupport \
--disable-lbmsupport \
--disable-tgasupport \
--disable-qpfsupport \
--disable-ttfsupport \
--disable-type1support \
--disable-latin9support \
--enable-gbsupport \
--enable-rbfgb12   \
--enable-vbfsupport \
--enable-fontcourier \
--enable-fontsserif  \
--enable-fontsymbol  \
--enable-fontvgas \
--disable-big5support \
--disable-unicodesupport \
--disable-pngsupport \
--disable-micemoveable \
--disable-cursor \
--enable-imegb2312 \
--enable-imegb2312pinyin \
--disable-savebitmap \
--enable-savescreen \
--enable-aboutdlg \
--disable-ext-fullgif \
--enable-flatstyle \
--disable-dblclk
make -j3
make install
```

MiniGUI1.3 以后的版本提供了和 Linux 内核图形配置界面一致的配置工具，使得 MiniGUI 的配置、交叉编译更加直观和方便，如图 11.19 所示。

当编译器和选型不在目前支持列表中时，我们需要修改 configs/config.in 和 Config ure.help 来添加支持，关于图形配置的详细资料请参考 MiniGUI1.3.x 的用户手册。

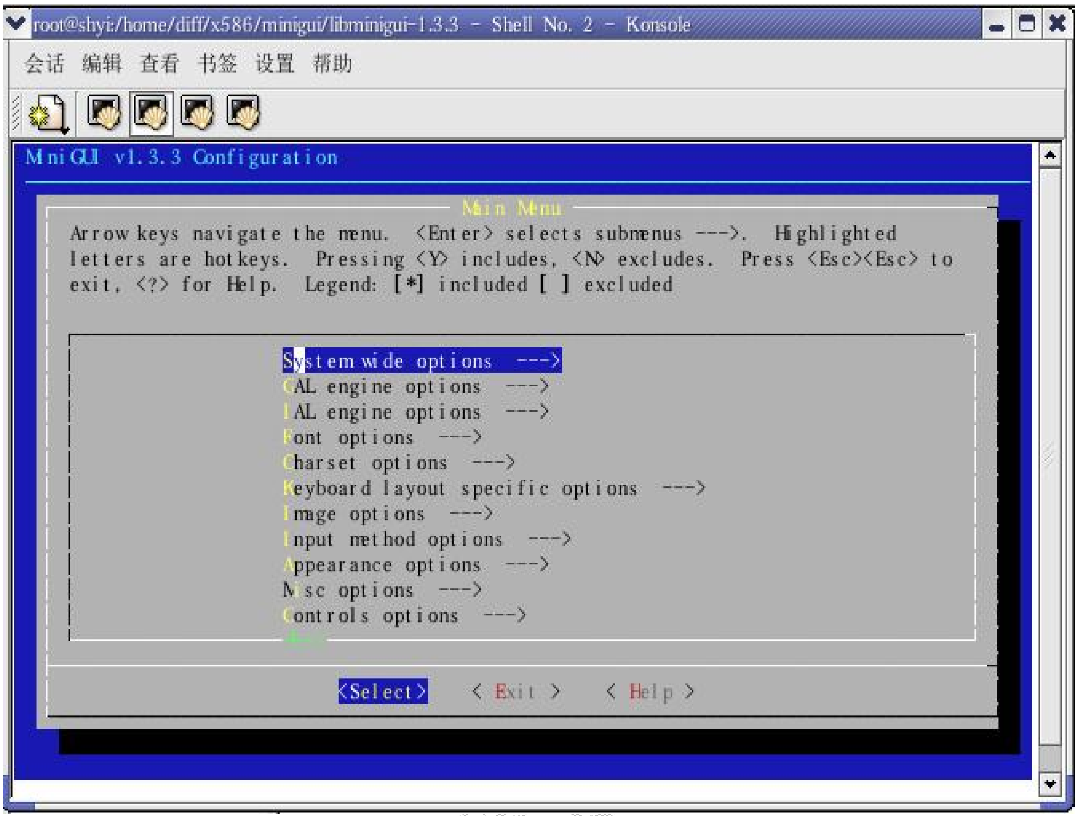


图 11.19 MiniGUI 编译图形配置界面

3. 安装资源文件

针对 MiniGUI1.3.x 版本发布的基本资源包，其中包含了运行 MiniGUI1.3.x 需要的基本字体（ISO8859-1）、鼠标光标、图标和位图等文件。另外还包括一款 UNICODE 编码的 UNIFONT QPF 字体，可用来显示 MiniGUI 所支持的所有字符集文字，包括 GB2312、BIG5、GBK 等。

```
# cd ../minigui-res-1.3.3/
```

下面的命令将使 TOPDIR 指向我们的目标系统根路径并安装到系统中。

```
vi config.linux
TOPDIR=$PREFIX
make install
```

4. 编译 MDE

MDE 是 MiniGUI 的综合演示程序。编译的时候由于系统中缺少了用于解析命令行选项的 popt.h 文件，从而导致 poptContext，POPT_ARG_INT，POPT_BADOPTION_NOALIAS 未定义。其相关的 MiniGUI 的 MDE 示范代码有/notebook/main.c，tools/vcongui.c、ontrolpanel/controlpanel.c、/controlpanel/panels.c，可以到下载一个源码包来安装到系

统中。

执行下面的指令来编译。

```
tar zxvf popt-1.7.tar.gz
cd poprt-1.7
make
make install
```

如果在编译时出现 vbf_Courier8x13 和 vbf_System14x16 未定义错误的提示，可以在 src/font/in-core/varbitmap.c 和 src/font/in-core/vbf_System14x16.c 文件中分别添加宏_INCOREFONT_COURIER 和_INCOREFONT_VGAS：

- 在 varbitmap.c 中添加如下内容。

```
#ifndef  _INCOREFONT_COURIER
#define _INCOREFONT_COURIER 1
#endif
#ifdef _VBF_SUPPORT
#ifdef _INCOREFONT_COURIER
```

- 在 vbf_System14x16.c 中添加如下内容。

```
#ifndef _INCOREFONT_VGAS
#define  _INCOREFONT_VGAS 1
#endif
#ifdef _VBF_SUPPORT
#ifdef _INCOREFONT_VGAS
```

配置 MDE 的参数之后就可以编译 MDE 了。

```
CC=arm-linux-gcc LDFLAGS=-L/$PREFIX/lib CPPFLAGS=-I/$PREFIX /include \
./configure \
--host=$TARGET \
--target=$TARGET \
--prefix=$PREFIX \
--exec-prefix=$PREFIX
```

编译安装还 MiniGUI 之后就可以做基于 MiniGUI 的开发了，下面通过一个简单的例子来介绍其编程使用方法。编写一个简单的 Hello，world！程序，程序清单如下。

```
#include <stdio.h>
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
```

```
#ifndef _LITE_VERSION
#include <minigui/dti.c>
#endif
static int HelloWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
     HDC hdc;
     switch (message) {
          case MSG_PAINT:
               hdc = BeginPaint (hWnd);
               TextOut (hdc, 100, 100, "Hello world!");
               EndPaint (hWnd, hdc);
               return 0;
          case MSG_CLOSE:
               DestroyMainWindow (hWnd);
               PostQuitMessage (hWnd);
               return 0;
     }
     return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

int MiniGUIMain (int argc, const char* argv[])
{
     MSG Msg;
     HWND hMainWnd;
     MAINWINCREATE CreateInfo;
#ifdef _LITE_VERSION
     SetDesktopRect(0, 0, 800, 600);
#endif
     CreateInfo.dwStyle = WS_VISIBLE | WS_BORDER | WS_CAPTION;
     CreateInfo.dwExStyle = WS_EX_NONE;
     CreateInfo.spCaption = "HelloWorld";
     CreateInfo.hMenu = 0;
     CreateInfo.hCursor = GetSystemCursor(0);
     CreateInfo.hIcon = 0;
     CreateInfo.MainWindowProc = HelloWinProc;
     CreateInfo.lx = 0;
     CreateInfo.ty = 0;
     CreateInfo.rx = 320;
     CreateInfo.by = 240;
```

```
        CreateInfo.iBkColor = COLOR_lightwhite;
        CreateInfo.dwAddData = 0;
        CreateInfo.hHosting = HWND_DESKTOP;

        hMainWnd = CreateMainWindow (&CreateInfo);
        if (hMainWnd == HWND_INVALID)
             return -1;
        ShowWindow(hMainWnd, SW_SHOWNORMAL);
        while (GetMessage(&Msg, hMainWnd)) {
             TranslateMessage(&Msg);
             DispatchMessage(&Msg);
        }
        MainWindowThreadCleanup (hMainWnd);
        return 0;
    }
```

该程序的运行结果如图 11.20 所示。

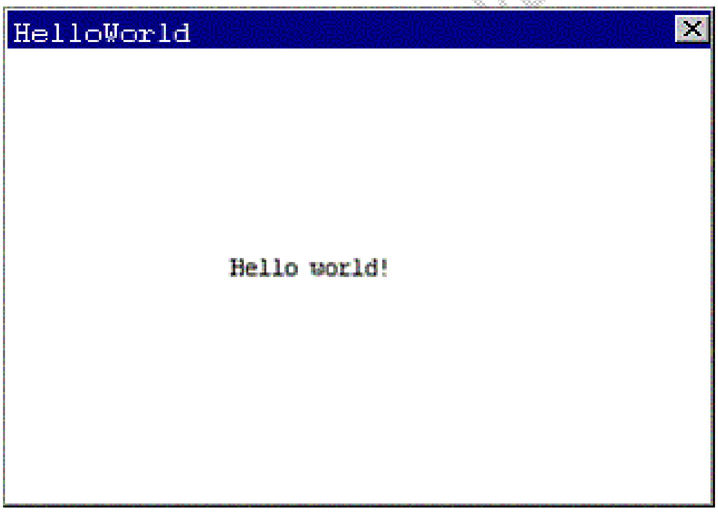


图 11.20　程序运行结果

MiniGUI 是一个典型的消息驱动的 GUI 系统，每个 MiniGUI 程序都是从 MiniGUIMain 函数开始的。应用程序先以 CreateMainWindow 函数创建一个主窗口，由于窗口创建的时候缺省是不可见的，我们通过 ShowWindow 函数显现该窗口，最后通过 GetMessage（&Msg，hMainWnd）进入消息循环状态。

11.6 MicroWindows 图形系统

MicroWindows 也是目前嵌入式系统开发图形系统采用较多的一种方案，对于大多数嵌入式设计，尤其是在运行专用图形程序的场合，X Window 不失一种恰当的选择。由此而发展起来的 MicroWindows 是专门设计用于在小型设备上开发具有高品质图形功能的开放式源码桌面系统。MicroWindows 目前由 Century 软件公司维护。它的主要特色在于提供了比较完善的图形功能，支持多种外部设备输入，包括液晶显示器、鼠标和键盘等。

Microwindows 起源于 NanoGUI 项目，它提供 2 种接口。

- Microsoft Windows Win32/WinCE 图形显示接口（GDI）
- Xlib-like 接口

前者应用于所有的 Windows CE 和 Win32 应用程序，后者就像 Nano-X，应用于所有 Linux X 插件集的最底层。这样做可以让大量的 Windows 程序员开发图形应用程序，类似地也可以让 Linux 图形程序员用 X 接口开发图形应用程序。

MicroWindows 基于 2.2.0 版本的 Linux 系统如图 11.21 所示。其内核所包含的代码允许用户程序将图形显示的内存空间作为 framebuffer 进行存取操作。这样在用户程序空间中可作为内存映射区域来直接控制图形显示，可使得用户在编写图形程序的时候不再需要去了解底层硬件，这也是目前 MicroWindows 在嵌入式系统中得到很多应用的原因所在。此外，MicroWindows 在运行过程中仅需要 50～250KB 的内存空间，远小于 X Windows 系统所需空间。这主要是因为 MicroWindows 对于在驱动层的每一个绘图函数采用的是单进程的方式，由驱动层核验是否裁减并调用驱动程序来绘制未被裁减的像素点或线，而在 X Window 系统中，则是出于对速度的考虑，包含所有像素点的绘制程序，并分别有裁减和未裁减的版本。

在嵌入式 Linux 平台上，从 Linux2.2.x 的内核开始，为了方便图形的显示，使用了 Framebuffer 技术。MicroWindows 可以运行在支持 32 位 Framebuffer 的 Linux 系统上，支持每个像素 1 位、2 位、4 位、8 位、16 位、24 位和 32 位的色彩空间/灰度，还实现了 VGA16 平面模式的支持，能通过调色板技术将 RGB 格式的颜色空间转换成目标机器上最相近的颜色，然后显示出来。

MicroWindows 采用分层设计方法。在其设计上有着明显的分层结构，其设备与平台相关层、设备与平台无关层和应用层之间层次清晰、结构明显，其层次结构图如图 11.22 所示。



图 11.21 基于 MicroWindows 的嵌入式 Linux 系统组成

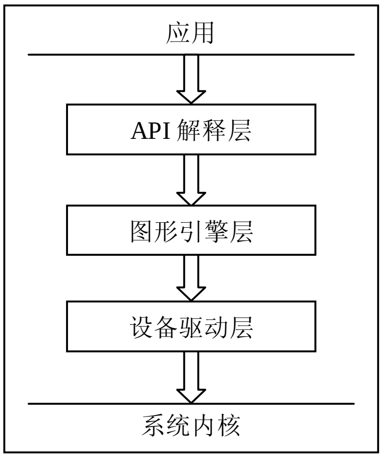


图 11.22 MicroWindows 系统分层结构

其中，在 MicroWindows 的最底层（设备驱动层），系统针对显示屏、鼠标、触屏以及键盘等各定义了一个数据结构。该数据结构和顶层结构一样，供图形引擎使用，包含了针对指定设置和硬件驱动代码。

Micro Windows 系统中的核心函数是在图形引擎层（也称为设备与平台无关层）通过调用下层硬件设备驱动程序来实现的。这些函数对屏幕、鼠标、键盘等驱动程序进行了封装，为 API 提供了服务，用户调用最上层提供的编程接口，而不需要直接调用引擎层的例程。

MicroWindows 的最高层（应用层）实现了窗口交互系统，这使得程序员既可在全屏幕显示，也可在层叠的窗口中显示。一层还包括事件处理，可将触摸屏、按键等激发的事件传送给应用程序。

MicroWindows 的核心基于显示设备接口，绝大部分是用 C 语言开发的，移植性很强。作为 X Window 图形系统的替代品，MicroWindows 可以使用更少的 RAM 和文件存储空间（100K～600K）提供与 X Window 相似的功能。目前已经移植到包括 ARM 在内的多种平台上。MicroWindows 有自己的 Framebuffer，因此它并不局限于 Linux 开发平台，比如在 eCos、FreeBSD、RTEMS 等其他操作系统上都能很好地运行。此外，MicroWindows 能在宿主机上仿真目标机，这给嵌入式开发提供了很大的便利。

11.7 Linux 下的网络应用

11.7.1 嵌入式设备的网络化

在网络应用日益普遍的今天，Internet 现已成为国际社会重要的基础信息设施，也是信息流通的重要渠道。将嵌入式系统能连接到 Internet 上面，则可以方便、低廉地将信息传送到几乎世界上的任何一个地方。嵌入式设备网络化的技术核心是在嵌入式系统中部分或完整地实现 TCP/IP 协议。TCP/IP 协议是一种目前被广泛采用的网络协议，由于在传统 8/16 位微控

制器上实现复杂的网络协议受内存和速度限制，而 TCP/IP 协议又比较复杂，所以实现起来是比较困难的，而现有嵌入式系统采用的处理器大多采用 32 位，甚至是 64 位的处理器，再加上操作系统的引用，这从技术实现上提供了很大的保障。Linux 天生就是一个优秀的网络操作系统，具有强大而且完备的网络特性，Linux 内核对网络协议栈的设计从简洁高效的角度出发，实现了一整套的网络协议模块。此外，使用开源免费的嵌入式 Linux 也有利于降低系统成本。

嵌入式系统实际上就是一个集成化的计算机系统。随着信息技术的发展，应用领域还对嵌入式系统提出了网络化功能，这就促使嵌入式系统向着更高的集成化方向发展。随着技术和市场的进一步发展，使得嵌入式系统可以与互联网实现通信，网络化的嵌入式设备使嵌入式系统功能更加强大，也更容易监控，也更有利于和其他网络设备实现数据交互。从数字技术和信息技术的角度看，嵌入式系统已成为现代信息网络技术应用的基础技术。

嵌入式设备的网络化应用有着广泛的应用前景，诸如在通信领域、信息家电、工业自动化等。相信在不久的将来，网络化的嵌入式设备将得到更加深入的研究与应用。

11.7.2　TCP/IP 协议概述

在当今的网络世界中，正在使用的网络协议主要有 2 种。

- OSI（Open System Interconnection Reference Model，由国际标准化组织 ISO 制定）
- TCP/IP（Translation Control Protocal/Intenet Protocal）

国际标准化组织 ISO 在 1977 年提出了 OSI 标准，该标准是一个 7 层通信协议的标准，其组成情况如图 11.23 所示。

应用层（Application）
表示层（Application）
会话层（Session）
传输层（Transport）
网络层（Network）
数据链路层（DataLink）
物理层（Physical）

图 11.23　OSI 7 层协议示意图

OSI 模型各层功能简单介绍如下。

① 应用层：即最接近用户的一层。一般是构筑在各种通讯协议上的网络应用软件，与用户直接交互。

② 表示层：表示层的功能是用标准的编码方式来统一数据格式，因为不同的计算机系统可能使用不同的数据编码方式，在这一层中要向应用层屏蔽这样的差异。另外，这一层还会完成对一些数据的处理加工。

③ 会话层：会话层用来控制传输连接时的数据交换，如传输方向、中断处理等。

④ 传输层：传输层使数据能正确无误地传输，控制数据流，TCP、UDP 协议属于该层。

⑤ 网络层：网络层在发送和接收之间建立一个虚拟的路径，即数据包从发送端到接收端的路由，IP 协议属于该层。

⑥ 数据链路层：数据链路层对下层传来的数据进行打包，将上层的数据分割成数据帧。

⑦ OSI 的物理层规范是有关传输介质的特性标准，这些规范通常也参考了其他组织制定的标准。连接头、针、针的使用、电流、电流、编码及光调制等都属于各种物理层规范中的内容。物理层常用多个规范完成对所有细节的定义。示例：Rj45，802.3 等。

在上面提到的两种协议标准中，使用最为广泛的无疑是 TCP/IP 协议，区别于 OSI 的 7 层结构，TCP/IP 采用的是 4 层的网络结构，如图 11.24 所示。

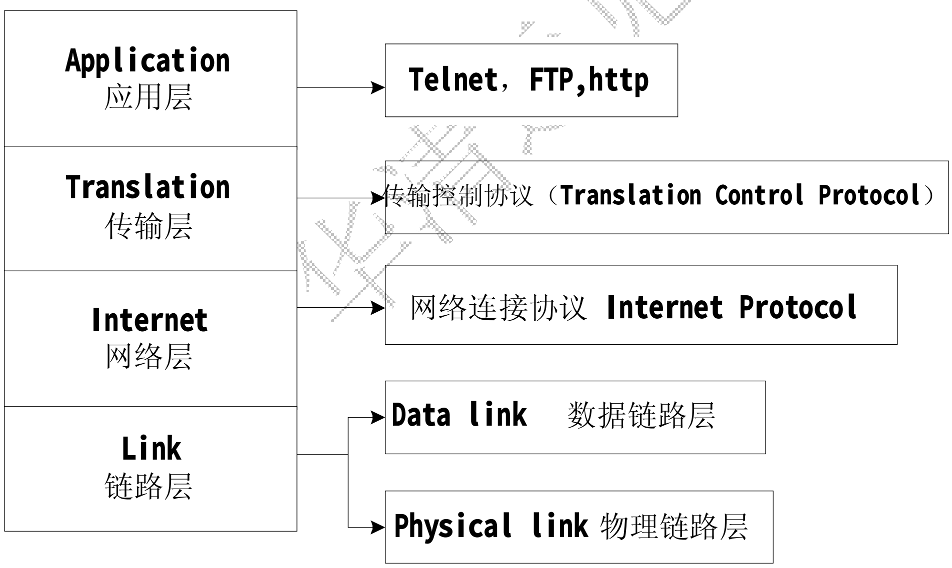


图 11.24 TCP/IP 协议模型

① 链路层，有时也称作网络接口层，通常包括操作系统中的设备驱动程序和计算机中对应的网络接口卡。它们一起处理与电缆（或其他任何传输媒介）的物理接口细节。

② 网络层，有时也称作互连网层，处理分组在网络中的活动，例如分组的路由选择。在 TCP/IP 协议组件中，网络层协议包括 IP 协议（网际协议），ICMP 协议（Internet 互连网控制报文协议），以及 IGMP 协议（Internet 组管理协议）。

③ 运输层主要为两台主机上的应用程序提供端到端的通信。在 TCP/IP 协议组件中，有两个实现点对点通讯的传输协议：TCP（传输控制协议）和 UDP（用户数据报协议）。

④ TCP（Transfer Control Protocol）传输控制协议是一种面向连接的协议，当我们的网络程序使用这个协议的时候，TCP 数据包中包括序号和确认，所以未按照顺序收到的包可以被排序，而损坏的包可以被重传，TCP 协议的数据包采用了比较复杂的格式来确保安全机制，所以基于 TCP 的传输可以保证客户端和服务端的连接是可靠的。

⑤ UDP（User Datagram Protocol）用户数据报协议是一种非面向连接的协议，这种协议并不能保证我们的网络程序的连接是可靠的，UDP 则为应用层提供一种非常简单的服务。它只是把称作数据报的分组从一台主机发送到另一台主机，但并不保证该数据报能到达另一端。但是 UDP 的一个显著优点是传输速度快，因此它应用于那些面向查询、应答的服务，例如 NFS 服务。

⑥ 应用层负责处理特定的应用程序细节。几乎各种不同的 TCP/IP 实现都会提供下面这些通用的应用程序。

- Telnet 远程登录
- FTP 文件传输协议
- SMTP 用于电子邮件的简单邮件传输协议
- SNMP 简单网络管理协议

TCP/IP 从一开始就集成到了 Linux 系统之中，并且其实现完全是重新编写的。现在，TCP/IP 已成为 Linux 系统中最健壮、速度最快和最可靠的部分，也是 Linux 系统之所以成功的一个关键因素。

11.7.3 Linux 下的 Socket 编程

Socket 是 TCP/IP 协议传输层所提供的接口（称为套接口），供用户编程访问网络资源，它是使用标准 Unix 文件描述符（file descriptor）和其他程序通信的方式。Linux 的套接口通信模式与日常生活中的电话通信非常类似，套接口代表通信线路中的端点，端点之间通过通信网络来相互联系。Socket 接口被广泛应用并成为事实上的工业标准。它是通过标准的 Unix 文件描述符和其他程序通信的一个方法。按其应用，套接口主要有以下两种分类。

- 流式套接字（Stream Socket）
- 数据报套接字（Datagram Socket）

小知识 tream Socket 采用 TCP 协议，而 Datagram Socket 采用 UDP 协议。

在 Linux 下使用套接口编程需要一系列的函数来操作，以下是 linux 下 socket 编程常用到的一些函数。

- socket（int domain，int type，int protocal）

头文件#include <sys/type.h>

#include <sys/socket.h>

socket()用来建立新的 Socket，也就是向系统注册，通知系统建立一个通讯端口，参数 domain()指定使用何种的地址类型，常见的协议有 UNIX 进程通信协议，IPv4 协议，IPv6 协议等；type 指定通信的方式，如 SOCK_STREAM（面向连接的 TCP 协议），SOCK_DGRAM

（面向无连接的 UDP 协议）。

注意 如果使用 TCP 协议，在所有数据传送之前必须使用 connect()来建立连线状态。

参数 protocal 用来指定 socket()所使用的传输协议编号，通常设定为 0 即可。

- bind（int sockfd，struct sockaddr *my_addr，int addrlen）

头文件#include <sys/type.h>

bind()函数用来设置给参数 sockfd 的一个名称，此名称由参数 my_addr 指向 sockaddr 结构，对于不同的 socket domain 定义了一个通用的数据结构。

```
Struct {
      unsigned  short int sa_family;
      char sa_data[14];
     }
```

参数 addrlen 为 sockaddr 的结构长度。

返回值：成功返回 0，失败则返回–1。

对于 bind()函数可以通过如下的赋值方法实现自动获取本机的 IP 地址和随机获取一个没有被占用的端口。

```
my_addr.sin_port=0;     /*系统随机选择一个未被使用的端口*/
my_addr.sin_addr.s_addr=INADDR_ANY;/*自动获取本机 IP 地址*/
```

- listen(int s,int backlog)等待连接函数

头文件#include <sys/socket.h>

listen()等待参数 s 的 socket 连线，用来监听是否有服务请求，如果希望等待一个进入的连接请求，然后再处理它们，就可以首先调用 listen()，然后再调用 accept()来实现，参数 backlog 同时能处理的最大连接要求，listen()并未开始接受连线，只是设置 socket 为 listen 模式，真正接受 client 端连接的是 accept()函数，通常 listen()会在 socket()，bind()，之后调用，接着才调用 accept()函数。

返回值：成功则返回 0，失败返回–1。

- accept（int s，struct sockaddr *addr int *addrlen）

头文件#include <sys/types.h>

#include <sys/socket.h>

accept()函数用来接受参数 S 的 socket 连接，参数 s 的 socket 必须先经 bind()、listen()函数处理过，当有连线进来时，accept()会返回一个新的 socket 处理代码，往后的数据传送与读取就是经由新的 socket()处理，而原来参数 s 的 socket 能继续使用 accept()来接受新的连线要求。连线成功时，参数 addr 所指的结构会被系统填入远程主机的地址数据，参数 addrlen 为 sockaddr 的结构长度。

返回值：成功则返回新的 socket 处理代码，失败返回–1。

- recv（int s,void *buf，int len，unsigned int flags）经 socket 接收数据

头文件#include <sys/types.h>

        #include <sys/socket.h>

recv()函数用来接收远端主机经指定的 Socket 传来的数据，并把数据存到由参数 buf 指向的内存空间，参数 len 为可接收数据的最大长度。

参数 flags 一般设定为 0。

返回值：若成功则返回接收到的字符数，失败返回–1。

- send（int s，const void *msg,int len, unsigned int flags）

头文件#include <systypes.h>

        #include <sys/socket.h>

send()函数用来将数据由指定的 Socket 传给对方主机，参数 s 为已建立好连线的 Socket（套接字描述符）。参数 msg 指向欲连线（欲发送）的数据内容，参数 len 则为该数据的数据长度。参数 flag 一般设为 0。

返回值：若成功则返回实际送出去的字符数的个数，失败则返回–1。

在使用 Socket 编程中需要注意的一个问题是，在计算机中，数据存储有两种字节优先顺序：高位字节优先和低位字节优先。在 Internet 上数据以高位字节优先的顺序在网络上传播，所以对于在内部以低字节优先方式存储数据的机器，所以在 Internet 上传输数据时需要进行转换。在 Socket 通讯中，用到以下 2 个结构体类型来实现以上转换，分别如下。

（1）struct sockaddr

```
{
unsigned  short  sa_family; /*地址族，AF_xxx*/
char  sa_data[14];/*14 字节的协议地址*/
}
```

sa_family 一般为 AF_INET，sa_data 则包含该 socket 的 IP 地址和端口号。

（2）struct　sockaddr_in

```
{
short  int  sin_family;       /*地址族*/
unsigned  short int  sin_port;   /*端口号*/
struct in_addr sin_addr;        /*IP 地址*/
unsigned  char sin__zero[8;] /*填充 0 以保持与 structsockaddr 同样大小*/
}
```

流式套接字提供了一种可靠的面向连接的数据传输方法，流式套接字需要由 socket()函数来创建，而调用时必须必须要用 bind()函数为它分配一个地址。图 11.25 是一个基于流式套接字的简单流程示意图。

在上面的流程图中，服务器端创建好套接字，并且赋给其一个地址之后，使用 listen()函数来侦听客户端的连接请求，如果连接成功就调用 accept()函数返回一个新的套接字描述符来处理双方的通信过程。客户端为了通知服务器端接收其发出的连接请求，也必须先建立一个 Socket，接着调用 connect()函数来发送连接请求。双方结束通信之后，都要调用 close()函数来结束 Socket 通信。基于上述流程，下面给出一个服务器/客户端的 Socket 通信例程。以下是服务器端的程序 comserv.c 清单。

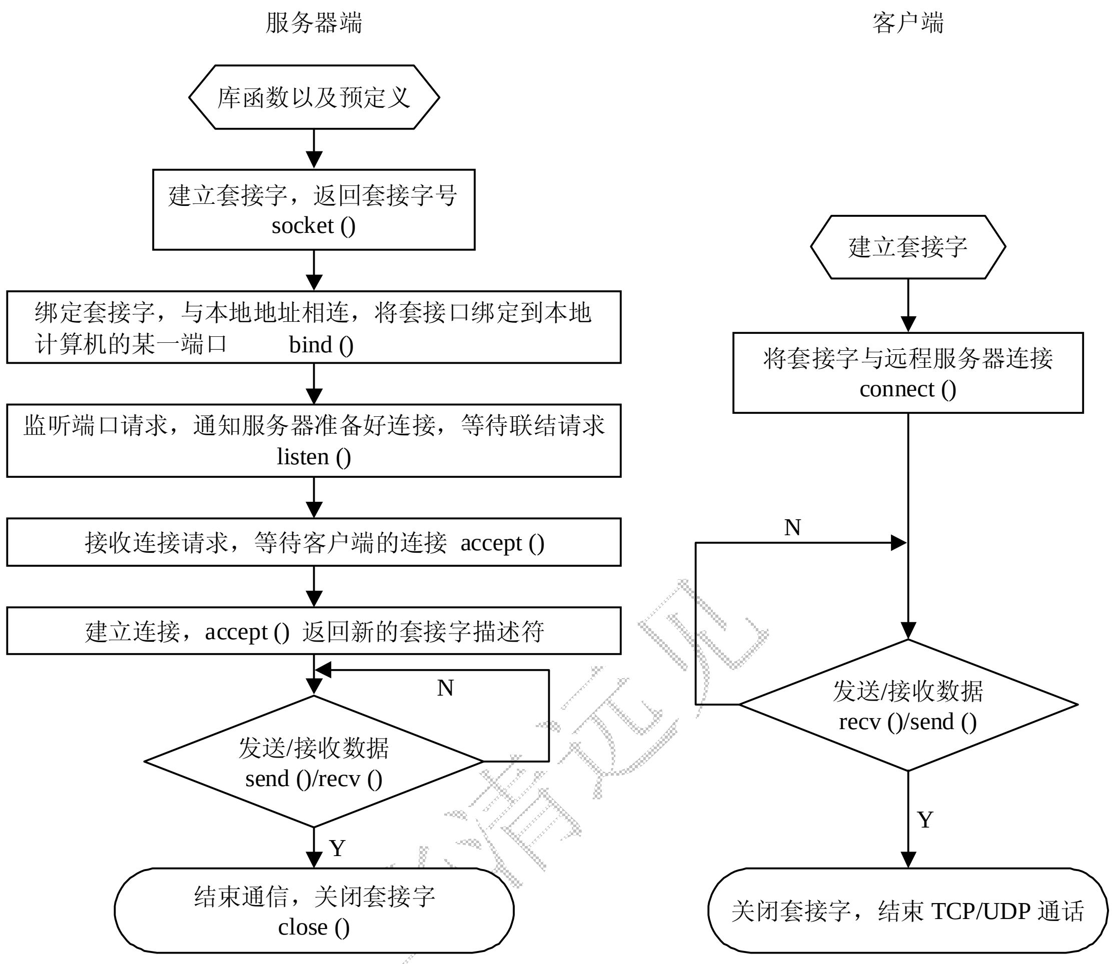


图 11.25 流式套接字流程示意图

```
/*****************Socket 通信服务器(Server)端程序 comserv.c:*****************/
/***************通信协议基于 TCP 协议***************/
#include<float.h>
#include<stdio.h>
#include<memory.h>
#include<unistd.h>
#include<signal.h>
#include<stdlib.h>
#include<errno.h>
#include<string.h>
#include<netdb.h>
#include<sys/time.h>
#include<sys/types.h>
```

```
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
/*定义全局变量*/

typedef struct ConnectData_tag
{
  int port;
  int waitForStart;
  int sFd;          /*监听\接收套接口描述符*/
  int msgFd;         /*读\写套接口描述符*/
}ConnectData;
ConnectData *CD;
int i=0;
int connectionMade=0;
/*功能模块*/
void prompt_info(int signumber)
{
  char src[ ]="Successed，this is a test for Socket Communication! \n";
                                                    /*定义字符串数组*/
  int nBytesToSet=strlen(src);                      /*返回字符串长度*/
  send(CD->msgFd,src,nBytesToSet,0);
}
void init_sigaction(void)                           /*定义信号处理函数*/
{
  struct sigaction act;
  act.sa_handler=prompt_info;
  act.sa_flags=0;
  sigemptyset(&act.sa_mask);
  sigaction(SIGPROF,&act,NULL);
}
void init_time(double t_usec)
{
  struct itimerval value;
  int int_usec=(int)(t_usec*1000000);
  value.it_value.tv_sec=0;
  value.it_value.tv_usec=int_usec;
  value.it_interval=value.it_value;
  setitimer(ITIMER_PROF,&value,NULL);
```

```
    }
    int ModeInit(void)
    {
      int error=0;
      error=ExtInit(CD);
      if(error!=0)goto EXIT_POINT;
      printf("Succeeded in creating listening Socket，Continue!！\n");
    EXIT_POINT:
      return (error);
    }
    int ExtInit(ConnectData *UD)
    {
      int sockStatus;
      int IPAddrStatus;
      struct sockaddr_in serverAddr;
      int sFdAddSize=sizeof(struct sockaddr_in);
      int option=1;
      int port=17725;
      int error=0;
      int sFd=-1;/*侦听套接口*/
      /*创建基于TCP的 socket*/
      memset((char *) &serverAddr,0,sFdAddSize);
      serverAddr.sin_family=AF_INET;
      serverAddr.sin_port=htons(port);
      serverAddr.sin_addr.s_addr=htonl(INADDR_ANY);/*转换为网络字节顺序*/
      sFd=socket(AF_INET,SOCK_STREAM,0);  /*创建套接字，AF_INET类型，使用TCP协议*/
      if(sFd==-1)
      {
        fprintf(stderr,"socket() call failed.\n");
        error=1;
        goto EXIT_POINT;
      }
      sockStatus=setsockopt(sFd,SOL_SOCKET,SO_REUSEADDR,(char *)&option, sizeof
(option));
      if(sockStatus==-1)
      {
        fprintf(stderr,"setsocketopt() call failed.\n");
        error=1;
        goto EXIT_POINT;
```

```
    }
    sockStatus=bind(sFd,(struct sockaddr *)&serverAddr,sFdAddSize);/*绑定套接
字于本地地址*/
    if(sockStatus==-1)
    {
      fprintf(stderr,"bind() call failed.\n");
      error=1;
      goto EXIT_POINT;
    }
    sockStatus=listen(sFd,1);/*侦听套接字请求*/
    if(sockStatus==-1)
    {
      fprintf(stderr,"listen() call failed.\n");
      error=1;
      goto EXIT_POINT;
    }
  EXIT_POINT:
    UD->msgFd=-1;
    UD->port=17725;
    if(error==1)
    {
      if(sFd!=-1)
      {
        close(sFd);          //关闭套接字
      }
      UD->sFd=-1;
    }
    else
    {
      UD->sFd=sFd;
    }
    return (error);
  }/*end ExtInit*/
  int OpenConnection(ConnectData *UD)
  {
    struct sockaddr_in clientAddr;
    int sFdAddSize=sizeof(struct sockaddr_in);
    int error=0;
    int  msgFd=-1;
```

```
    const int  sFd=UD->sFd;
    msgFd=accept(sFd,(struct sockaddr *)&clientAddr,&sFdAddSize);  /*等待接收客
户端请求*/
    if(msgFd==-1)
    {
      fprintf(stderr,"accept() for message socket failed.\n");
      error=1;
      goto EXIT_POINT;
    }
    printf("server:get connection from %s\n",inet_ntoa(clientAddr.sin_addr));
    connectionMade=1;
  EXIT_POINT:
    if(error!=0)
    {
      if(msgFd!=-1)
      {
        close(msgFd);
      }
      UD->msgFd=-1;
    }
    else
    {
      UD->msgFd=msgFd;
      if(msgFd!=-1)
      {
        printf("Succeeded in creating socket!\n");
      }
    }
    return (error);
  }
  /*主程序*/
  int main(int argc,const char *argv[ ])/*main 函数的参数决定了执行时命令行的格式*/
  {
    int error;
    const char *option=argv[1];
    CD=(ConnectData *)malloc(sizeof(ConnectData));
    memset(CD,0,sizeof(ConnectData));
    if(strcmp(option,"-E")==0)
    {
```

```
    CD->waitForStart=1;
  }
  else
  {
    CD->waitForStart=0;
  }
  ModeInit();
  while((CD->waitForStart)&&(connectionMade==0))
  {
    error=OpenConnection(CD);
    if(error)
    {
      exit(1);
    }
  }
  init_sigaction();
  init_time(1.0);     /*每隔一秒发送一次传输内容*/
  while(1);
  close(CD->msgFd);
  sleep(1);
  close(CD->sFd);*/
  return (0);
}/*end main*/
/*********************************结束***********************************/
```

以下是客户端的程序 comclient.c 清单。

```
/*********************客户端(Client)通信程序 comclient.c:******************/
#include<float.h>
#include<stdio.h>
#include<memory.h>
#include<unistd.h>
#include<signal.h>
#include<stdlib.h>
#include<errno.h>
#include<string.h>
#include<netdb.h>
#include<sys/time.h>
#include<sys/types.h>
#include<sys/socket.h>
```

```
#include<netinet/in.h>
#include<arpa/inet.h>
/*typedef int SOCKET;*/
#define MAXSIZE 50
#define PORT 17725
static void bail(const char *on_what)
{
    fputs(strerror(errno),stderr);
    fputs(";",stderr);
    fputs(on_what,stderr);
    fputc('\n',stderr);
    exit(1);
}
int main(int argc,char **argv)
{
    int sockfd;
    int z;
    int numbytes;
    int IPAddrStatus;
    int i;
    /*char *srvr_addr=NULL;*/
    char buf[MAXSIZE];
    struct sockaddr_in adr_srvr;
    int addr_port=PORT;
    int len_inet=sizeof(struct sockaddr_in);
    memset(&adr_srvr,0,len_inet);
    adr_srvr.sin_family=AF_INET;
    adr_srvr.sin_port=htons(addr_port);
    IPAddrStatus=inet_aton("172.25.22.174",&adr_srvr.sin_addr); /*此处可以填
入相应服务器端的 IP 地址*/
    if(IPAddrStatus==0)
        bail("inet_aton()");
    sockfd=socket(AF_INET,SOCK_STREAM,0);
    if(sockfd==-1)
        bail("socket()");
    z=connect(sockfd,(struct sockaddr *)&adr_srvr,len_inet);
    if(z==-1)
        bail("connect()");
    for( i=0;i<5;i++)
```

```
    {
      numbytes=recv(sockfd,buf,MAXSIZE,0);
      if(numbytes==-1)
         bail("recv()");
      buf[numbytes]='\0';
      printf("Received:%s",buf);    /*将接收的内容重复 5 次打印*/
    }
    close(sockfd);  /*关闭客户连接*/
    return 0;
  }
  实际编译运行过程:
  [root@Localhost  zxq]# gcc –o sr  comserv.c
  [root@Localhost  zxq]# gcc –o clt comclient.c
  [root@Localhost  zxq]# ./s1 -E &
  [1] 5684
  Succeeded in creating listening Socket，Continue! !
  [root@ Localhost  zxq]# ./clt
  server:get connection from 172.25.22.174
  Succeeded in creating socket!
  Received: Successed，this is a test for Socket Communication!
  Received: Successed，this is a test for Socket Communication!
  Received: Successed，this is a test for Socket Communication!
  Received: Successed，this is a test for Socket Communication!
  Received: Successed，this is a test for Socket Communication!
  [root@ Localhost  zxq]#

  /*********************************结束***********************************/
```

11.8 嵌入式 Linux 的串行通信

11.8.1 Linux 下的串口操作

串行口是计算机一种常用的接口，因为具有连接线少，通信简单的特点，得到广泛的使用。常用的串口是 RS-232-C 接口（又称 EIA RS-232-C）。它是在 1970 年由美国电子工业协会（EIA）联合贝尔系统、调制解调器厂家及计算机终端生产厂家共同制定的用于串行通信的标准。串口通信指的是计算机依次以位（bit）为单位来传送数据，串行通信使用的范围很

广，在嵌入式系统开发过程中串口通信也经常用到通信方式之一。

Linux 对所有设备的访问是通过设备文件来进行的，串口也是这样。为了访问串口，只需打开其设备文件即可操作串口设备。在 Linux 系统下，每一个串口设备都有设备文件与其关联，设备文件位于系统的/dev 目录下面。如 Linux 下的/ttyS0，/ttyS1 分别表示的是串口 1 和串口 2。下面详细介绍 Linux 下是如何使用串口的。

1. 串口操作需要用到的头文件

```
#include    <stdio.h>      /*标准输入输出定义*/
#include    <stdlib.h>     /*标准函数库定义*/
#include    <unistd.h>     /*Unix 标准函数定义*/
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <fcntl.h>      /*文件控制定义*/
#include    <termios.h>    /*POSIX 终端控制定义*/
#include    <errno.h>      /*错误号定义*/
#include    <string.h>     /*字符串功能函数*/
```

2. 串口通信波特率设置

波特率的设置定义在<asm/termbits.h>，其包含在头文件<termios.h>里。

常用的波特率常数如下。

B0------- **à** 0　　B1800------- **à** 1800
B50----- **à** 50　　B2400------ **à** 2400
B75----- **à** 75　　B4800------ **à** 4800
B110---- **à** 110　　B9600------ **à** 9600
B134---- **à** 134.5　　B19200----- **à** 19200
B200---- **à** 200　　B38400------ **à** 38400
B300---- **à** 300　　B57600------ **à** 57600
B600---- **à** 600　　B76800------ **à** 76800
B1200--- **à** 1200　　B115200----- **à** 115200

假定程序中想要设置通信的波特率，一般使用 cfsetispeed()和 cfsetospeed()函数来操作，获取波特率信息是通过 cfgetispeed()和 cfgetospeed()函数来完成的。比如可以如下指定串口通信的波特率。

```
#include <stdio.h>    //头文件定义
........
........
.......
struct termios opt;          /*定义指向 termios 结构类型的指针 opt*/
```

```
/***************以下设置通信波特率****************/
cfsetispeed(&opt，B9600 );  /*指定输入波特率，9600bps*/
cfsetospeed(&opt，B9600); /*指定输出波特率，9600bps*/
/************************************************/
.........
..........
```

一般来说，输入、输出的波特率应该是一致的。

3．串口属性配置

在程序中，很容易配置串口的属性，这些属性定义在结构体 struct termios 中。为在程序中使用该结构体，需要包含文件<termbits.h>，该头文件定义了结构体 struct termios。该结构体定义如下。

```
#define NCCS 19
struct termios {
                tcflag_t c_iflag;            /* 输入参数 */
                tcflag_t c_oflag;            /* 输出参数 */
                tcflag_t c_cflag;            /* 控制参数*/
                tcflag_t c_ispeed;           /* 输入波特率 */
                tcflag_t c_ospeed;           /* 输出波特率 */
                cc_t c_line;                 /* 线控制 */
                cc_t c_cc[NCCS];             /* 控制字符*/
              };
```

其中成员 c_line 在 POSIX(Portable Operating System Interface for Unix)系统中不使用。

对于支持 POSIX 终端接口的系统中，对于端口属性的设置和获取要用到以下 2 个重要的函数。

（1）int tcsetattr（int fd，int opt_DE，*ptr）

该函数用来设置终端控制属性，其参数说明如下。

- fd 待操作的文件描述符
- opt_DE：选项值，有 3 个选项以供选择

TCSANOW：不等数据传输完毕就立即改变属性。

TCSADRAIN：等待所有数据传输结束才改变属性。

TCSAFLUSH：清空输入输出缓冲区才改变属性。

- *ptr 指向 termios 结构的指针

函数返回值：成功返回 0，失败返回–1。

（2）int tcgetattr（int fd，*ptr）

该函数用来获取终端控制属性，它把串口的默认设置赋给了 termios 数据数据结构，其参数说明如下。

- fd 待操作的文件描述符

- *ptr 指向 termios 结构的指针

函数返回值：成功返回 0，失败返回–1。

4．打开串口

在前面已经提到 Linux 下的串口访问是以设备文件形式进行的，所以打开串口也即是打开文件的操作。函数原型可以如下所示。

```
int open（“DE_name”，int open_Status）
```

参数说明：

（1）DE_name：要打开的设备文件名。

比如要打开串口 1，即为/dev/ttyS0。

（2）open_Status：文件打开方式，可采用下面的文件打开模式。

- O_RDONLY：以只读方式打开文件。
- O_WRONLY：以只写方式打开文件。
- O_RDWR：以读写方式打开文件。
- O_APPEND：写入数据时添加到文件末尾。
- O_CREATE：如果文件不存在则产生该文件，使用该标志需要设置访问权限位 mode_t。
- O_EXCL：指定该标志，并且指定了 O_CREATE 标志，如果打开的文件存在则会产生一个错误。
- O_TRUNC：如果文件存在并且成功以写或者只写方式打开，则清除文件所有内容，使得文件长度变为 0。
- O_NOCTTY：如果打开的是一个终端设备，这个程序不会成为对应这个端口的控制终端，如果没有该标志，任何一个输入，例如键盘中止信号等，都将影响进程。
- O_NONBLOCK：该标志与早期使用的 O_NDELAY 标志作用差不多。程序不关心 DCD 信号线的状态，如果指定该标志，进程将一直在休眠状态，直到 DCD 信号线为 0。

函数返回值：成功返回文件描述符，如果失败返回–1。

例如假定以可读写方式打开/dev/ttyS0 设备，就可以如下操作。

```
#include<stdio.h>    //头文件包含
......
......
int fd; /* 文件描述符 */
fd = open("/dev/ttyS0", O_RDWR | O_NOCTTY);   /*以读写方式打开设备*/
if(fd == -1)
perror("Can not open Serial_Port 1\n! "); /*打开失败时的错误提示*/
........
........
```

5．串口读操作（接收端）

用 open 函数打开设备文件，函数返回一个文件描述符（file descriptors，fd），通过文件描述符来访问文件。读串口操作是通过 read 函数来完成的。函数原型如下。

```
int read(int fd, *buffer,length);
```

参数说明：
（1）int fd：文件描述符；
（2）*buffer：数据缓冲区；
（3）length：要读取的字节数。
函数返回值：读操作成功读取返回读取的字节数，失败则返回–1。

6．串口写操作（发送端）

写串口操作是通过 write 函数来完成的。函数原型如下。

```
write(int fd, *buffer,length);
```

参数说明：
（1）fd：文件描述符；
（2）*buffer：存储写入数据的数据缓冲区；
（3）length：写入缓冲去的数据字节数。
函数返回值：成功返回写入数据的字节数，该值通常等于 length，如果写入失败返回–1。
例如：向终端设备发送初始化命令

```
#include<stdio.h>    //头文件包含
......
......


int n
sbuf[]={Hello，this is a Serial_Port test! \n }; //待发送数据
int len_send=sizeof（sbuf）; //发送缓冲区字节数定义
n = write(fd,Sbuf,len_send); //写缓冲区
if(n == -1)
{
printf("Wirte sbuf error.\n");
}
......
......
```

7．关闭串口

对设备文件的操作与对普通文件的操作一样，打开操作之后还需要关闭，关闭串口用函数 close()来操作，函数原型如下。

```
int close(int fd);
```

参数说明：
fd：文件描述符。
函数返回值：成功返回 0，失败返回–1。

11.8.2 Linux 串口编程实例

为了说明问题，我们举一个简单的例子来理解上一节提到的串口操作流程，例程 receive.c 用来接收从串口发来的数据，而例程 send.c 用来发送数据到串口。二者成功建立串口连接后，串口接收端会收到串口发送端发来的字符串数据“Hello，this is a Serial Port test!”。

1．receive.c 程序清单

```
/*******************************************************
*ilename: receive.c
* Description: Receive data from Serial_Port
* Date:
*******************************************************/
/*********************头文件定义***********************/
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <termios.h>
#include "math.h"
#define max_buffer_size   100   /*定义缓冲区最大宽度*/
/*********************************************************/
int fd, s;
int open_serial(int k)
{
   if(k==0)      /*串口选择*/
   {
```

```
        fd = open("/dev/ttyS0",O_RDWR|O_NOCTTY);  /*读写方式打开串口*/
        perror("open /dev/ttyS0");
    }
   else
    {
        fd = open("/dev/ttyS1",O_RDWR|O_NOCTTY);
        perror("open /dev/ttyS1");
    }
   if(fd == -1)  /*打开失败*/
    return -1;
   else
    return 0;
}
/*********************************************************************/
int main()
{
char  hd[max_buffer_size],*rbuf; /*定义接收缓冲区*/
int flag_close, retv,i,ncount=0;
struct termios opt;
int realdata=0;
/********************************************************************/
open_serial(0);    /*打开串口1*/
/********************************************************************/
tcgetattr(fd,&opt);
cfmakeraw(&opt);
/******************************************************************/
cfsetispeed(&opt,B9600); /*波特率设置为9600bps*/
cfsetospeed(&opt,B9600);
/********************************************************************/
tcsetattr(fd,TCSANOW,&opt);
rbuf=hd; /*数据保存*/
printf("ready for receiving data...\n");
retv=read(fd,rbuf,1);   /*接收数据*/
if(retv==-1)
{
 perror("read"); /*读状态标志判断*/
}
/************************开始接收数据******************************/
while(*rbuf!='\n')      /*判断数据是否接收完毕*/
```

```
 {
     ncount+=1;
     rbuf++;
     retv=read(fd,rbuf,1);
     if(retv==-1)
     {
perror("read");
     }
 }
/*******************************************************************/
printf("The data received is:\n");  /*输出接收到的数据*/
for(i=0;i<ncount;i++)
{
      printf("%c",hd[i]);
}
printf("\n");
flag_close =close(fd);
if(flag_close = = -1)   /*判断是否成功关闭文件*/
printf("Close the Device failur! \n");
return 0;
}
/****************************结束**********************************/
```

2. send.c 程序清单

```
/*******************************************************
* File Name:     send.c
* Description:   send data to serial_Port
* Date:
*******************************************************/
/******************头文件定义******************/
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <termios.h>
```

```
#define max_buffer_size   100   /*定义缓冲区最大宽度*/
/*******************************************/
int fd;  /*定义设备文件描述符*/
int flag_close;
int open_serial(int k)
{
   if(k==0)      /*串口选择*/
     {
        fd = open("/dev/ttyS0",O_RDWR|O_NOCTTY);  /*读写方式打开串口*/
        perror("open /dev/ttyS0");
    }
   else
    {
        fd = open("/dev/ttyS1",O_RDWR|O_NOCTTY);
        perror("open /dev/ttyS1");
    }
   if(fd == -1)  /*打开失败*/
     return -1;
   else
    return 0;
}
/********************************************************************/

int main(int argc, char *argv[])
{
 char sbuf[]={"Hello,this is a Serial_Port test!\n"};/*待发送的内容，以\n 为结
束标志*/
 int sfd,retv,i;
 struct termios option;
 int length=sizeof(sbuf);/*发送缓冲区数据宽度*/
/*******************************************************************/
open_serial(0);    /*打开串口 1*/
/*******************************************************************/
printf("ready for sending data...\n"); /*准备开始发送数据*/
tcgetattr(fd,&option);
cfmakeraw(&option);
/*****************************************************************/
cfsetispeed(&opt,B9600); /*波特率设置为 9600bps*/
cfsetospeed(&opt,B9600);
```

```
/*******************************************************************/
tcsetattr(fd,TCSANOW,&option);
retv=write(fd,sbuf,length);  /*接收数据*/
if(retv==-1)
{
 perror("write");
}
printf("the number of char sent is %d\n",retv);

flag_close =close(fd);
if(flag_close = = -1)   /*判断是否成功关闭文件*/
printf("Close the Device failur! \n");

return 0;
}
/****************************结束**********************************/
```

分别将上面的两个程序编译之后就可以运行了，如果是在两个不同的平台上运行，比如，在开发板上运行数据发送程序 write（write.c 编译后得到），在宿主机上运行接收数据程序 read（read.c 编译得到），采用串口线将二者正确连接之后，就可以运行来看实际的效果了。

首先在宿主机端运行数据接收程序 receive。

```
[zhang@localhost]# ./receive
[zhang@localhost]#open /dev/ttyS0:  Success
    ready for receiving data...
    The data received is:
    Hello,this is a Serial_Port test!
[zhang@localhost]#
```

在接收端运行完程序之后再到发送端运行数据发送程序 send。

```
#./send
ready for sending data...
the number of char sent is 35
#
```

运行完发送程序之后就可以在接收端看到接收的数据了。

也可以在一台 PC 机上来运行这两个程序，这时将串口线的 2、3 脚短路连接即可（自发自收），实际运行的步骤与上面相同。

“黑色经典”系列之《嵌入式 Linux 系统开发技术详解——基于 ARM》

FARSIGHT

第 12 章　系统集成测试

本章目标

本章介绍了系统集成测试需要的各种工具，主要包括系统跟踪、性能测试和内存测试 3 个方面。通过学习本章内容，可以了解一些基本的系统测试方法，用来解决系统性能方面的问题。

系统跟踪工具
系统性能测试
系统内存测试

12.1 系统集成测试

Linux 系统的内核、应用程序和文件系统分别来源于不同的软件工程，当把它们都集成到一个系统中时，可能会出现意想不到问题。解决这些问题需要一些测试过程。

12.1.1 系统集成测试概述

当内核、应用程序等各部分组成 Linux 系统以后，系统的状况就会变得复杂起来。因为系统中有大量程序同时执行，任务的调度、系统资源的管理都有可能出现问题。例如：一个任务长时间得不到运行；可用内存越来越少；系统越来越慢等等。

对于这样的问题，只有把系统组件都集成到一起才能够发现，还要借助一些调试测试工具才能解决。如同处理内核的 BUG 一样，必须先重现这些问题，才能更好地解决问题。那么先来建立集成测试环境。

在交叉开发环境下，我们已经开发完成了 Linux 内核、应用程序和文件系统。先不要急着去裁减文件系统。如果把辅助测试工具和调试信息去掉了，将很难对系统状态进行分析。通过 NFS 文件系统，可以方便地把这些系统组件和工具集成到一个测试环境中，而不必担心文件系统太大。

集成测试环境与交叉开发环境的建立步骤是基本相同的，不同的是已经包含了新开发的内核、应用程序以及文件系统配置。测试环境建立的前提是已经完成所有内核和应用程序开发，仍然采用 NFS 文件系统方式，添加测试程序和配置文件。

在这种测试环境下，大部分应用程序都是可以完整测试的。

12.1.2 系统集成测试要求

对于嵌入式系统，需要测试的指标可能很多。从 Linux 操作系统角度来说，主要包括下列 3 个方面的测试。

- 系统功能测试

检查测试系统的功能，通常是应用程序执行的结果。这对于复杂的系统软件是必要的。

- 系统性能测试

可以在长时间、高负载等条件下测试系统性能，甚至可以模拟极限情况。

- 内存泄漏测试

如果程序有内存泄漏，系统的空闲内存会越来越少。测试内存泄漏就是要找出程序内存泄漏的位置。内存测试也可以说是性能测试，这里把内存测试列为一个单独的重要项目。

如同 Linux 使用 GNU 的开发工具一样，Linux 的大部分测试工具都是开发源码的软件工具。但是这些软件工具通常只在 X86 的平台上开发并使用，或者仅支持少数几种体系结构。

对于嵌入式 Linux 开发来说，移植是永恒的话题。任何开放源码的程序都可以做跨平台的移植工作。测试工具也需要相应地移植工作，包括内核和工具的程序修改。对于 Linux 发行版，Linux 公司会整合一套有效的测试工具，提供给客户使用。例如：Montavista Linux 产品包含了 strace、mtrace、LTT 等工具包。

另外，有些商业的测试工具也可以支持 Linux 开发测试，这些开发工具对开发工作将非

常有用。

12.2　系统跟踪工具

12.2.1　为什么需要跟踪工具

当系统运行的时候，有时通过打印信息根本无法判断程序出错的原因。例如：系统突然崩溃等。

GDB 调试器适合调试源代码，可以解决一些程序错误。但是它不能有效地解决应用程序之间或者应用程序与内核之间的交互。这类问题必须跟踪应用程序与其他软件之间的交互信息，根据实际交互信息分析出错原因。

可以通过系统跟踪工具来解决这些问题。最基本的跟踪方式是检测一个应用程序与 Linux 内核之间的数据交互。这样就容易观察到传递的参数或者系统调用顺序，分析出错的原因。

然而，孤立地观察一个进程并不能满足所有的情况。如果试图调试进程间的同步问题或者时间顺序严格的问题，需要一个系统范围的跟踪机制，提供系统的确切时序和事件发生的时间。

系统与用户的交互有各种各样的方式，主要有系统调用、信号处理、动态库调用等。每一种方式都有一种或者几种测试跟踪工具。

12.2.2　Strace

Strace 是单个 Linux 进程的跟踪工具，它能够跟踪并且打印出程序调用的所有系统调用。它使用 ptrace 系统调用跟踪调试运行中的进程。它不要求重新编译要跟踪的程序，即使没有源代码，同样可以调试跟踪。

系统调用和信号是发生在用户和内核接口的事件，跟踪它们对于查找 BUG 是非常有用的。这正是 Strace 能够解决的问题。

Strace 最初是为 SunOS 系统编写的，现在已被移植到了大部分 Unix 系统和 Linux 系统中。Strace 工具软件遵守 BSD 软件许可，从下列站点可以获取最新的版本。

```
http://www.liacs.nl/~wichert/strace/
```

安装 Strace 既可以采用本地编译方式，也可以采用交叉编译方式。以 Strace 版本 4.5.14 为例说明编译安装过程。

下载软件包并且解压。

```
$ tar –jxvf strace-4.5.14.tar.bz2
$ cd strace-4.5.14
```

（1）在目标板上本地编译安装

这和在工作站上编译安装 Strace 的过程一样。通过 configure 命令可以猜测目标机的系统类型，调用本地的 GCC 编译。

```
$ ./configure
$ make
```

（2）在开发主机上交叉编译安装

在 configure 配置过程中需要通过 CC 定义交叉编译工具链，并且需要指定目标机。例如：编译 arm-linux 平台的 Strace。

```
$ CC=arm-linux-gcc ./configure --host=arm-linux
$ make
```

二进制程序编译完成以后，复制到目标板的根文件系统中。

```
$ cp strace /usr/local/arm/3.3.2/rootfs/usr/sbin
```

然后在目标板的 Linux Shell 就可以使用 Strace 命令了。例如：

```
# strace -f -o ls.strace ls
```

上面的命令就是跟踪 ls 及其子进程的运行，将输出信息写到文件 ls.strace 中，然后就可以分析 ls.trace 文件中的信息了。

12.2.3　Ltrace

Ltrace 工具也是单个 Linux 进程的跟踪工具。Ltrace 与 Strace 跟踪的对象不同，它跟踪的是动态库函数的调用；两者使用的方法基本相同，都不需要重新编译程序。

Ltrace 最早出现在 GNU/Debian Linux 中，现在至少已经支持 Debian、RedHat、SuSE 和 Mandrake 等 Linux。Montavista Linux 软件也包含 Ltrace。

Ltrace 的源代码可以从下列站点获取最新版本。

```
http://ltrace.alioth.debian.org
```

值得注意的是 BUG 修正的列表，它的功能也在不断完善中，将来会支持更多的 Linux 体系结构和版本。

Ltrace 的安装过程与 strace 相同，可以支持本地编译和交叉编译，这里不再重复说明。

使用 Ltrace 命令可以跟踪运行中的一个进程。例如：

```
# ltrace -p 234
```

上面命令就是通过 Ltrace 跟踪一个 pid 为 234 的进程，可以打印出进程运行时所有的动态库函数调用信息。

12.2.4　LTT

LTT（Linux Trace Toolkit）是 Linux 重要的系统跟踪工具。

1．LTT 工具的特点

LTT 通过一个内核模块来监测主要的内核子系统。内核的跟踪模块采集产生的数据，转发给用户空间的守候进程并且记录到磁盘上。

整个过程对系统运行和性能影响很小。许多测试已经证明：这个跟踪系统在不使用的时候，影响几乎可以忽略；即使在一些压力条件下，影响也小于 2.5%。

LTT 还提供了事件观测工具，用 3 种不同的格式来分析跟踪的数据（事件图、进程分析、原始事件）。LTT 对于系统性能分析是很有用的；对于获取实时或者非实时的任务在内核和用户层面的交互信息也非常有用。它的主要功能如下。

- 调试进程间的同步问题。
- 分析应用程序和内核之间的交互。
- 分析系统对外部输入事件的响应。
- 测量内核为应用程序提供服务的执行时间。
- 测量进程等待较高优先权进程的时间。
- 测量中断处理时间和对系统的影响。

因此，LTT 软件工具也比较复杂，一般可以分成 3 个部分：内核模块、数据保存和数据分析工具。

2．LTT 软件介绍

LTT 是基于 GPL 发布的自由软件，它是 Karim Yaghmour 创建并维护的。网址为：

```
http://www.opersys.com/ltt/index.html
```

最近正式发布的版本是 0.9.5a。对于 Linux 2.6 内核，需要使用 0.9.6 以上版本，可以使用 ltt-0.9.6-pre4.tar.bz2 软件包。

软件源代码是按照目录组织分类的，表 12.1 是主要目录的说明。

表 12.1　　LTT 源码目录说明

目　　录	说　　明
Daemon	跟踪进程（Trace Daemon）的源代码
Examples	各种例子
ExtraScripts	方便 LTT 使用的脚本
Help	包含 HTML 帮助文件的目录
LibLTT	包含 LTT 事件数据库的目录
LibUserTrace	包含用户跟踪库的目录
Patches	包含不同内核补丁的目录
Visualizer	可视化分析工具的源码目录
其余目录	其余文件包括 autoconf/automake 包，可以简化或者自动编译 LTT 软件。 另外，“Example”目录的编译是由它自己的 Makefile 独立编译的

要使用 LTT 工具，先仔细阅读一下“help”目录下的帮助文档。

通常 LTT 工具需要特定 Linux 内核版本的补丁，各种体系结构 Linux 的支持也不同。Montavista Linux 在所有的产品中包含了一套完整的 LTT 软件工具。

3．安装软件工具

安装 LTT 工具之前要选择适当版本的 LTT 软件包。对于 Linux 2.4 内核比较简单，正式发布的软件包都包含了必要的补丁。对于 Linux 2.6 内核，需要 0.9.6 以后的版本才能支持。

Linux 2.6 内核的 LTT 建立过程也有变化。为了使能 LTT 和 relayfs，必须首先在源代码上打补丁。这些补丁修改代码中相关的地方，然后内核配置界面下使能跟踪支持选项和 relayfs 文件系统选项。

LTT 在 2.6 内核上采用了 relayfs 文件系统。relayfs 用于从内核空间向用户空间高效地转移数据。安装跟踪工具之后，需要先挂接 relayfs，然后在给定时间内执行跟踪进程。最新的 Linux 2.6 内核版本已经采纳了 relayfs 文件系统。

下载 LTT 软件包并且解压，编译安装 LTT 的 3 个部分。

```
$ tar  -jxvf  ltt-0.9.6-pre4.tar.bz2
```

（1）编译安装新内核

要让内核产生跟踪信息，必须修补内核。在 Patches 目录下有以下两个补丁。

```
$ ls ltt-0.9.6-pre4/Patches
ltt-linux-2.6.9-vanilla-041214-2.2.patch
relayfs-2.6.9-041124.patch
```

这 2 个补丁分别对应 LTT 和 relayfs 的支持。然而，因为内核不断发展，所以经常需要更新内核补丁。通常可以到 http://www.opersys.com/ftp/pub/LTT/ExtraPatches/取得新版内核补丁。

如果使用的是不同的内核，可以试着按照补丁手工修改内核。最新的 Linux 2.6 内核已经支持 relayfs，不再需要 relayfs 的补丁。

修改完内核以后，就可以配置编译内核了。选择“Linux Trace Toolkit support”菜单为“Y”。在 LTT 0.9.6pre2 之前发行的补丁中，可以选择为模块，以动态方式加载跟踪驱动程序。之后的版本完全作为一个子系统实现而不是设备驱动。

内核编译安装过程很简单。这个内核选项在系统开发完成以后就可以去掉了，但是建议保留这个可跟踪的内核。将来可以用于跟踪系统现场运行的问题，实际上跟踪系统造成的系统花销很小。

（2）编译安装跟踪监控程序

跟踪监控程序负责将数据写入永久性存储设备。存储设备可以是磁盘或者 MTD 设备，开发环境下最好是 NFS 文件系统。跟踪时间越长，存储数据量越大。

在 LTT 源码目录下编译安装跟踪调试程序。将 LDFLAGS 的值设置成-static，这样会生成 LibUserTrace 静态链接库。静态链接可以避免在目标板上再安装额外的库，程序的可移植性好。对于 C 库仍然使用动态链接方式，不然程序尺寸将大幅度增加。

编译完成之后，将跟踪监控程序以及跟踪辅助命令脚本复制到目标板的根文件系统。

trace 命令脚本是启动跟踪监控程序的最简单方法。也可以直接使用 tracedaemon 工具，命令参数要复杂一些。

（3）安装可视化工具

可视化工具安装运行在主机上，负责数据的分析显示。它即支持命令行方式，也支持图形方式。图形界面无疑是最直观的数据分析方法。如果准备使用图形接口，系统上必须安装 GTK。缺省的情况下，大多数 Linux 主机系统都会安装 GTK。如果希望通过命令脚本的分析跟踪数据，就要把它当成命令行工具使用。

编译源代码，得到可视化工具 tracevisualizer，把它和辅助命令脚本安装到主机文件系统目录下，配置相应的路径。

4．LTT 的使用和结果分析

现在可以使用 LTT 工具跟踪目标板的 Linux 系统了。目标板最好采用 NFS 文件系统，这对于测试数据的保存和传递都是很方便的。如果觉得 NFS 方式的网络流量可能影响跟踪数据，可以先用 TMPFS 文件系统来保存数据，跟踪完成后再复制过来。

执行下面的命令，启动跟踪目标板 30s。

```
# trace 30 out
```

命令参数 out 指定了存储文件名。命令执行完毕会产生 2 个文件：out.trace（包含原始的二进制跟踪数据）和 out.proc（包含跟踪开始时系统状态的快照）。然后在主机上使用可视化工具查看分析这两个文件的数据。

主机端可以直接运行 tracevisualizer，然后菜单打开跟踪数据。也可以使用如下命令行来检查跟踪数据，弹出图 12.1 所示可视化工具窗口。

```
$ traceview out
```

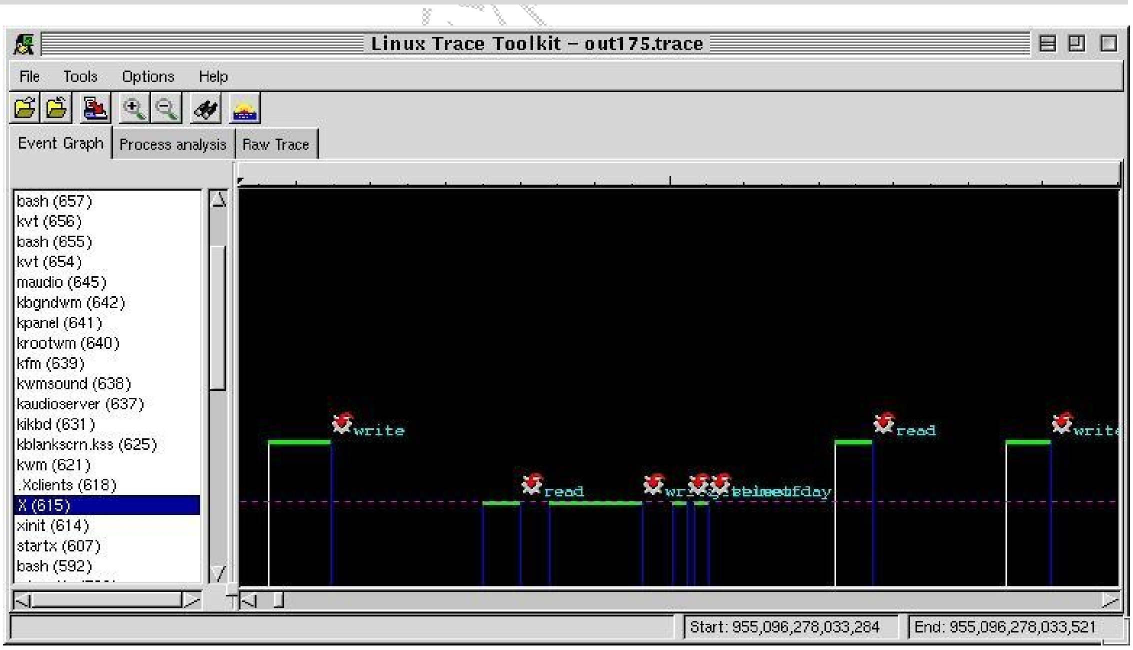


图 12.1　LTT 可视化工具的图形界面

在可视化工具左边窗格列出跟踪期间所有活动的进程树，Linux 内核永远处于最下边。

在可视化工具的右边窗格是描述系统运行过程的图形。图形的横坐标代表时间，纵坐标代表 Linux 任务或者内核运行状态。随着时间向前走，系统在不同的任务和内核之间切换。当发生系统调用或者中断的时候，纵坐标对应内核态。

这个“Event Graph”图形可以横向放大或者缩小，还可以使用滚动条向左右移动来检查某一段事件的变化。

另外还有“Process Analysis”（图 12.2 所示）和“Raw Trace”（图 12.3 所示）界面，使用这种图形，可以轻易找出应用程序与系统的其他进程之间的交互。

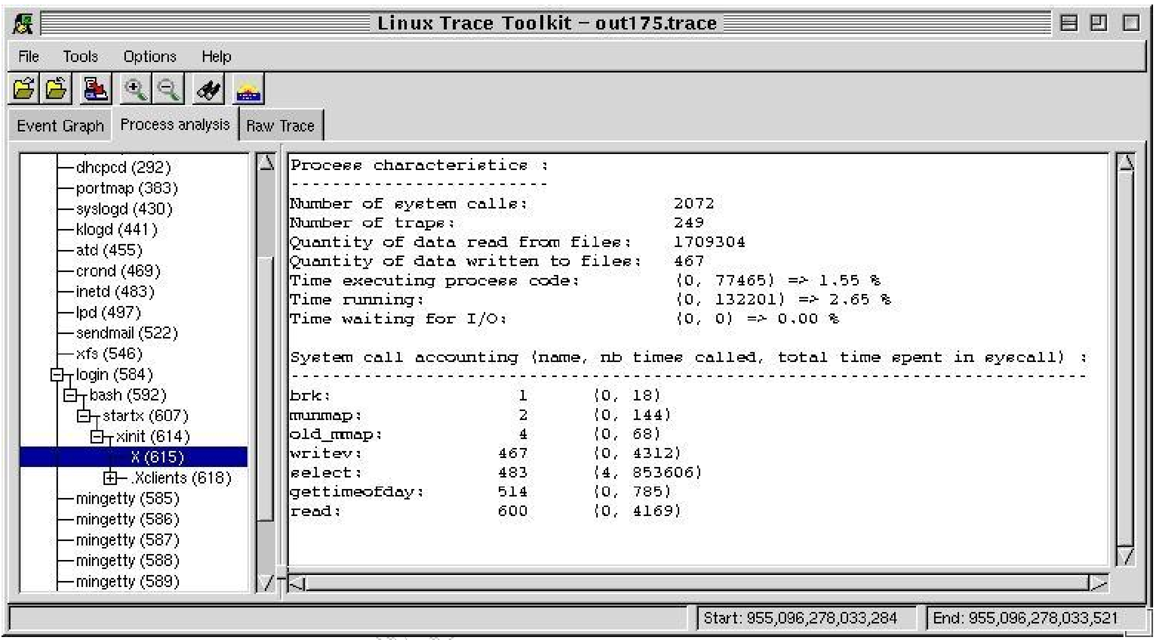


图 12.2 LTT 可视化工具的进程分析界面

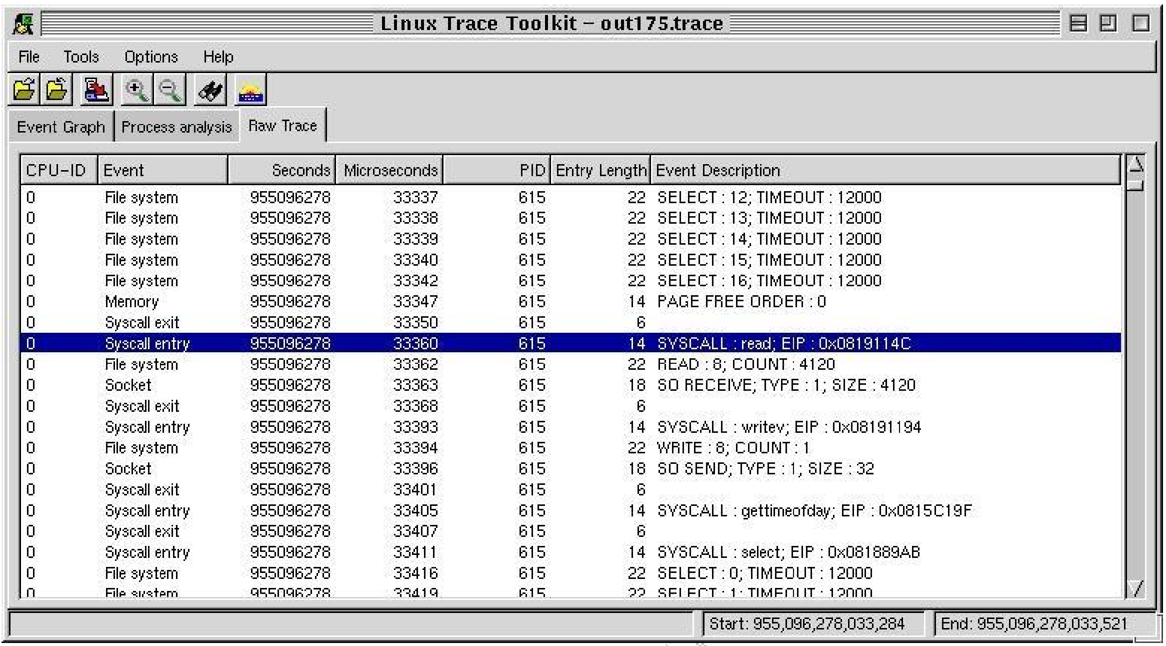


图 12.3　LTT 可视化工具的原始数据界面

如果需要命令行方式，可以把 tracevisualizer 当成命令行工具使用。这时 tracevisualizer 命令会读取 2 个输入文件，产生一个原始事件清单文本文件。这份清单就是“Raw Trace”功能显示的内容。

```
$ tracevisualizer  out.trace  out.proc  out.data
```

out.trace 和 out.proc 是采集的数据文件，跟踪数据会转储到 out.data 文件中。也可以使用 tracedump 或者 traceanalyzer 之类的命令脚本分析。

12.3　系统性能测量工具

对于 Linux 服务器来说，系统性能是衡量产品的一个重要标志。由于嵌入式 Linux 系统的处理器是千差万别的，系统的性能也不可能有统一的指标。然而，Linux 的各种性能测量工具可以用来优化调整嵌入式 Linux 系统的特定性能。

12.3.1　代码效率测量

1．gprof

gprof 是代码执行测试工具，它可以测量程序中函数执行所花的时间，还能计算代表其他进程运行的时间。

使用 gprof 工具还需要一个特殊的编译器选项。通过这个编译选项编译源程序之后，在

程序运行时会收集统计数据，并且在应用程序结束时保存成文件。然后，才可以使用 gprof 工具分析统计这些数据。

首先，必须修改应用程序的 Makefile，加入下列的编译器选项和链接器选项。

```
CFLAGS  = -Wall –pg
LDFLAGS     = -pg
```

编译器标志和链接器标志都应该包含“-pg”选项。这个“-pg”选项告诉编译器在所编译的源代码中纳入产生性能数据的程序代码；告诉链接器用 gcrtl.o（而不是 crtl.o）来链接二进制程序。另外，不要使用“-O2”优化选项，让生成的应用程序会确实按照源码文件指定的方式进行。

重新编译好应用程序之后，把它放到目标板上执行。尽量让应用程序充分执行，让所有的程序代码都可能被操作到。应用程序结束执行之后，会产生一个内含统计数据的输出文件 gmon.out。然后，可以使用主机端的 gprof 工具来分析结果。对于交叉编译的程序，要使用代前缀的交叉 gprof 工具。把 gmon.out 文件复制到应用程序的源代码目录下，执行下列命令分析 test 程序。

```
$ gprof  hello
```

结果从标准输出打印函数调用统计数据。通常使用“>”重定向到一个文件中保存。命令操作不需要特别指定 gmon.out 文件，因为它是自动加载的。关于 gprof 用法的更多信息请参考 GNU 的 gprof 手册。

2. gcov

gcov 是代码覆盖测试工具，它可以分析程序源代码行的调用次数，看其中哪些频繁调用，哪些没有调用过。

gcov 的功能需要工具链的支持，因为它是在编译 GCC 编译器的时候一起被编译进 libgcc 链接库的。GCC3.0 以前的版本的交叉编译器就不支持代码覆盖测试的功能。其 libgcc 不会包含可以产生程序代码覆盖范围相关的适当程序代码。所以，除非修改 GCC 源码，否则无法进行程序的代码覆盖测试。GCC3.0 以后的版本可以支持代码测试功能。

要支持 gcov 工具，还需要添加编译器选项。

```
CFLAGS   = -Wall –fprofile-arcs –ftest-coverage
```

同样也要去掉“-O2”优化选项，这样才能得到真正与原始代码相对应的“代码覆盖”数据。

通过“–fprofile-arcs –ftest-coverage”编译选项，编译源程序时将生成两个文件：*.bb 和 *.bbg。然后在目标板上执行程序，结果为每一个源码文件产生一个.da 文件。

最后，通过 gcov 工具生成测试结果。

```
$ gcov hello.c
```

结果生成.gcov 文件。这个文件是包含“覆盖范围”信息的程序文本文件。关于 gcov 的

使用或产生的输出的更多信息，可参考 GNU GCC 使用手册的“gcov”部分。

3．Oprofile

Oprofile 是一种代码评测和性能监控工具。

Oprofile 工具包含内核模块和用户空间守护进程两部分。内核模块可以访问性能计数寄存器，用户空间的守护进程负责从这些寄存器中收集数据。在启动守护进程之前，Oprofile 需要配置事件类型以及每种事件的样本计数（sample count）。Oprofile 被设计成可以在低开销下运行，从而使后台运行的守护进程不会扰乱系统性能。

Oprofile 内核模块已经包含在 Linux 2.6 版本的内核中，可以工作在多种体系结构上。要使用 Oprofile，需要重新配置编译内核。在内核配置菜单使能“Profiling support”选项，结果在.config 文件中应该有下列两行。

```
CONFIG_PROFILING=y
CONFIG_OPROFILE=y
```

接下来配置安装 Oprofile 工具。从下列站点下载软件包。

```
http://oprofile.sourceforge.net
```

解压后配置编译安装。

```
#./configure --with-kernel-support
#make
#make install
```

这样就得到了 Oprofile 的工具集。其中，opcontrol 就是 Oprofile 的守护进程控制工具。opcontro 具体的命令参数说明见表 12.2。

表 12.2　　opcontrol 命令行选项说明

opcontrol 选项	说　　明
--list-events	列出处理器事件和单元屏蔽（unit mask）
--vmlinux=<kernel image>	要分析的内核映像文件
--no-vmlinux	不分析内核
--reset	清除已经采集的数据
--setup	在守护进程启动之前的设置
--event=<processor event>	监视给定的处理器事件
--start	开始数据采样
--dump	把数据读到守护进程中
--stop	停止数据采样
-h	关闭守护进程

另外还有 opreport 和 opannotate 工具，用于分析收集到的数据。

Oprofile 评测的过程是在目标板本地运行过程中进行的。下面是基本测试流程。

（1）配置 Oprofile

```
# opcontrol --setup --vmlinux=/boot/vmlinux
```

（2）清除采集数据

```
# opcontrol --reset
```

（3）启动评测

```
# opcontrol --start
```

Oprofile 开始工作，这时可以在 Linux Shell 下执行一些要评测的应用程序。

（4）停止评测

```
# opcontrol --stop
```

（5）获取评测数据

```
# opreport > output_file
```

最后，使用 opannotate 工具和--source 选项或者--assembly 选项生成报告文件。报告中将包含程序代码的样本数和事件统计结果。

12.3.2 LTP

LTP（Linux Test Project）是 SGI、IBM、OSDL 和 Bull 合作的项目，目的是为开源社区提供一个测试套件，用来验证 Linux 系统可靠性、健壮性和稳定性。LTP 是 Linux 内核和相关特点测试的一整套工具集，它力求通过自动化的测试方法改进 Linux 内核。

LTP 是基于 GPL 的软件工程，从下列站点可以免费下载源码包。

```
http://ltp.sf.net
```

LTP 站点提供完整的源码包，也有功能分离的测试包。这里介绍一下完整的源码包：ltp-full-20060306.tgz。解压就可以得到完整的 LTP 工具源代码，目录结构见表 12.3 中的说明。

表 12.3 LTP 源代码说明

doc/*	工程文档包含工具和库函数使用手册，描述各种测试
include/*	通用的头文件目录
lib/*	通用的库函数目录
pan/*	包含一个简单的、轻便的测试装置。“Pan”具备随机和并行测试的能力
testcases/*	包含所有运行在 LTP 下的测试程序和链接
testscripts/*	存放分组的测试脚本
tools/*	存放自动化测试脚本和辅助工具
runtest/*	为自动测试提供命令列表

scratch/*	存放零碎测试的东西，还没有集成到 LTP 工程中
Makefile	LTP 顶层目录的 Makefile，负责编译安装 pan、testcases 和 tools
runalltests.sh	可以顺序运行全部测试例程并且报告结果的脚本
runltplite.sh	可以配置进行部分测试的脚本

LTP 提供了大量的测试工具和脚本，基本的测试流程如下。

（1）以 root 用户身份登录。

（2）解压 ltp 软件包。

通常放在用户的目录下或者工作目录下。

```
# tar -xzf ltp-full-20060306.tgz
# cd ltp-full-20060306
```

（3）编译安装。

编译过程很简单，执行 make 命令即可。但是有些 Makefile（例如：pan/Makefile）需要 lex 或者 flex 的支持。

```
# make
```

安装时会创建 testcases/bin 目录，然后把各种测试工具链接到这个目录下。

```
# make install
```

（4）运行 runalltests.sh 脚本，顺序执行全部测试。

运行脚本简单，但是这个测试过程极为复杂，正常情况下也要运行很长时间。对于嵌入式 Linux 系统来说，可以使用 runltplite.sh 脚本，仅测试特定的几个方面。

（5）按照项目分别测试。

可以通过 testscripts 目录下的脚本分别测试 Linux 各子系统。例如：diskio.sh 可以专门测试磁盘 I/O 文件系统，networktests.sh 可以测试网络性能，还有网络压力测试 networkstress.sh 等。

现在的 LTP 测试包提供了几千个测试用例，覆盖内核的大多数接口，例如：系统调用、内存、IPC、I/O、文件系统和网络等。测试套件每月都会更新发布，可以运行于多种体系结构上。LTP 测试套件已经能够支持 i386、PowerPC、S/390、MIPS 以及各种嵌入式体系结构等。通过 LTP 可以对 Linux 系统进行压力和性能测试。

12.3.3 LMbench

LMbench 是一种 benchmark（性能测试，后面将直接引用英文单词）软件，针对各种通用系统应用而设计。多数情况下，用来测试系统实际性能问题，并且常用于比较不同系统的实现。有些情况下，benchmark 可以发现一些新的 BUG 和设计缺陷。LMbench 包含一个可扩展的测试结果数据库。

LMbench 最早是由 Larry McVoy 创建的，后来 Carl Staelin 也加入支持和开发工作。它是基于 GPL 自由发布的。从 Sourceforge 站点可以下载 LMbench 软件包。

```
http://jaist.dl.sourceforge.net/sourceforge/lmbench/lmbench-3.0-a4.tgz
```

LMbech 提供了一套 benchmark，可以测量大多数相当大范围系统应用的常见性能瓶颈。这些瓶颈已经被识别，隔离和重现，通过一套微型 benchmark，可以测量系统延迟和处理器和内存、网络、文件系统、磁盘之间的数据传递的带宽，目的是产生实际应用程序能够重现的数量，而不是经常报告并且不时重现的市场性能问题。

benchmark 关注延迟和带宽，因为性能问题通常是由于延迟问题、带宽问题或者两者结合所导致的。每一个 benchmark 可以捕捉一些独特的性能问题，在一个或者多个重要应用程序中体现。例如：TCP 延迟 benchmark 可以精确预测 Oracle 的锁管理的性能，内存延迟 benchmark 给出一个强烈的 Verilog 仿真性能提示，并且文件系统延迟 benchmark 建立了软件开发中关键路径的模型。

在 1993～1995 年，LMbench 是在许多机器上为识别和评估系统性能瓶颈开发的。完全有可能因为未来计算机体系结构的变化或者更加高级，而废弃 benchmark。对于嵌入式 Linux 系统来说，反而更加需要。

LMbench 已经在最终用户和系统设计方面广泛应用。在一些情况下，LMbench 已经提供了数据发现和修正性能问题的必要数据。LMbench 在 Sun 的内存管理软件中发现了一个问题：使所有的页面映射到 cache 的同一个地方，高效地把 512KB cache 转换成 4KB cache。

lenbench 只测量系统的在处理器、cache、内存、网络和磁盘之间传输数据的能力。它不能测量系统的其他部分，例如：图形子系统，也不是 MIPS（百万指令每秒）、MFLOPS、吞吐率、饱和度、压力、图形或者多处理器的测试软件。经常在多处理器的系统运行，用来和单处理器比较性能问题，但是不能利用多处理器的特点。

因为是使用标准的可移植的系统接口和应用程序常用工具写的，所以 LMbench 是在广泛的 Unix/Linux 系统上可移植和可对比的。LMbench 已经运行在 AIX、BSDI、HP-UX、IRIX、Linux、FreeBSD、NetBSD、OSF/1、Solaris 和 SunOS 上。部分测试工具已经运行在 Windows 上。

12.4 测量内存泄漏

对于嵌入式系统来说，内存是非常宝贵的资源。

作为 Linux 系统的标准编程语言，C 语言对动态内存分配有很大的控制权。这反而可能会导致严重的内存管理问题，这些问题可能导致程序崩溃或导致系统性能不断下降。

内存泄漏和缓冲区溢出是一些常见的问题，它们可能很难检测到。这一部分将讨论几个调试工具，它们极大地简化了检测和找出内存问题的过程。

12.4.1 mtrace

mtrace 是最简单的一种内存泄漏跟踪工具。

mtrace 可以探测由于不成对使用 malloc/free 函数调用引起的内存泄漏。通过 GNU C 库的函数调用 mtrace()实现，可以打开跟踪并且创建分配和释放地址的日志文件。通过一个 perl 脚本（也叫作 mtrace）显示日志文件，列出不成对的 malloc()发生的源代码行号。在 Linux 下可以用于检查 C 和 C++程序。mtrace 具有可伸缩性的特点，它可以用来做全面的

程序调试。

使用 mtrace 有 3 个方面最关键。

（1）包含 mcheck.h；

（2）设置 MALLOC_TRACE 环境变量；

（3）调用 mtrace()函数。

如果没有设置 MALLOC_TRACE 变量，mtrace()不会执行任何操作。

mtrace 输出包含一些信息，例如：

```
- 0x0804a0f8 Free 13 was never alloc'd
/memory_leak/memory_leaks/mtrace/my_test.c:193
```

说明了释放的内存从来没有被分配，内存没有释放的内存段包含了 malloc 调用的地址、大小和行号。

12.4.2 dmalloc

dmalloc（Debug Malloc Library）是替代 malloc、realloc、calloc、free 和其他内存管理函数的库。

dmalloc 的特点提供了内存泄漏跟踪和越界写检测功能。它可以报告出错的程序文件名、行号和一些通用的统计信息。它在运行时具有可配置性。dmalloc 库是 Gary Watson 维护的，已经移植到 Linux 等许多操作系统上。

dmalloc 可以配置包含线程支持和 C++支持。它能够作为共享库和静态库编译。所有这些选项在编译的时候选择，在链接应用程序的时候用到的这些库。有一个头文件 dmalloc.h 需要包含在应用程序中。除了库和头文件，必须设置一个 dmalloc 读取的环境变量，用来配置如何检查和在哪里保存日志信息。dmalloc 测试程序使用的设置命令如下。

```
$ export DMALLOC_OPTIONS=debug=0x44a40503,inter=1,log=logfile
```

这行命令的意思如下。

- log 是当前目录下的一个文件 logfile；
- inter 是库检查自己的频率为 1；
- debug 是选择检查类型的 16 进制数。

把每一个可能的错误都定义在测试列表中，每个错误对应“debug”的一个位。下列是各种错误类型的定义。

```
none (nil): no functionality (0)
log-stats (lst): log general statistics (0x1)
log-non-free (lnf): log non-freed pointers (0x2)
log-known (lkn): log only known non-freed (0x4)
log-trans (ltr): log memory transactions (0x8)
log-admin (lad): log administrative info (0x20)
log-blocks (lbl): log blocks when heap-map (0x40)
log-bad-space (lbs): dump space from bad pointers (0x100)
```

```
    log-nonfree-space (lns): dump space from non-freed pointers (0x200)
    log-elapsed-time (let): log elapsed time for allocated pointer (0x40000)
    log-current-time (lct): log current time for allocated pointer (0x80000)
    check-fence (cfe): check fencepost errors (0x400)
    check-heap (che): check heap adm structs (0x800)
    check-lists (cli): check free lists (0x1000)
    check-blank (cbl): check mem overwritten by alloc-blank, free-blank (0x2000)
    check-funcs (cfu): check functions (0x4000)
    force-linear (fli): force heap-space to be linear (0x10000)
    catch-signals (csi): shut down program on SIGHUP, SIGINT, SIGTERM (0x20000)
    realloc-copy (rco): copy all re-allocations (0x100000)
    free-blank (fbl): overwrite freed memory space with BLANK_CHAR (0x200000)
    error-abort (eab): abort immediately on error (0x400000)
    alloc-blank (abl): overwrite newly alloced memory with BLANK_CHAR (0x800000)
    heap-check-map (hcm): log heap-map on heap-check (0x1000000)
    print-messages (pme): write messages to stderr (0x2000000)
    catch-null (cnu): abort if no memory available (0x4000000)
    never-reuse (nre): never reuse freed memory (0x8000000)
    allow-free-null (afn): allow the frees of NULL pointers (0x20000000)
    error-dump (edu): dump core on error and then continue (0x40000000)
```

如果库需要检查 C++程序，还需要一个名为 dmalloc.c 的源文件。这个模块提供封装函数，把 new 转换成 malloc，把 delete 转换成 free。GNU 调试器 GDB 可以结合 dmalloc 使用，可以配置.gdbinit 文件，以便让 GDB 自动配置 dmalloc。

这个库随带的工具是 dmalloc，它会配置 DMALLOC_OPTIONS 变量。这里已经做了一个脚本，在运行调试程序之前生效。

更多 dmalloc 的源代码和文档可以参考以下网站。

```
    http://dmalloc.com/
```

12.4.3 memwatch

memwatch 是一种 C 语言内存错误检测工具。它是由 Johan Lindh 编写的，开放源代码 memwatch 不仅能够探测 malloc 和 free 错误，而且能够探测越界（fencepost）情况。当向一块分配的内存（通过 malloc 分配）并且数据超越分配区域的末端时，会发生越界的情况。也有 memwatch 不能捕捉到的情况：把数据写到已经释放的地址并且从外部分配的内存读取数据。

memwatch 的核心是 memwatch.c 文件。它实现了封装和地址检查的代码。使用 memwatch 前要做好以下准备。

- 必须在源码中包含 memwatch.h 头文件。

- 必须在编译命令行定义变量 MEMWATCH（-DMEMWATCH）和 MW_STDIO（-DMW_STDIO）。
- 必须和应用程序同时使用 memwatch.c 文件，memwatch.c 生成的目标模块必须链接到应用程序中。

对于应用程序的执行，如果 memwatch 发现任何不正常，就会在标准输出上打印一行信息。创建文件 memwatch.log 文件存储出错信息，每一个错误信息包含出错的行号和源文件名。

memwatch 支持 ANSI C，它提供结果日志记录，能检测双重释放（double-free）、错误释放（erroneous free）、没有释放的内存（unfreed memory）、溢出和下溢等等。

查看 memwatch.log 的日志，在内存地址改变导致分配区域的起始和结尾的重叠地方，memwatch 工具能发现越界（fencepost）的情况。memwatch 缺点是不能定制，它必须运行在整个应用程序上。

举例说明 memwatch 的使用方法。test.c 是引起内存泄漏的一个例程。

```
/*  test.c */
#include "memwatch.h"
int main(void)
{
   char *ptr1;
   char *ptr2;
   ptr1 = malloc(512);
   ptr2 = malloc(512);
   ptr2 = ptr1;
   free(ptr2);
   free(ptr1);
}
```

test.c 程序分配了两个 512B 的内存块，然后使 2 个指针都指向第一块内存。第二内存地址丢失，无法正常释放申请的第二块内存，结果产生了内存泄漏。

通过 memwatch 来测试这个问题。

```
 # gcc -DMEMWATCH -DMW_STDIO  -o test test.c memwatch.c
```

运行 test 程序，结果会生成一个报告 memwatch.log。

```
MEMWATCH 2.67 Copyright (C) 1992-1999 Johan Lindh
...
double-free: <4> test.c(15), 0x80517b4 was freed from test.c(14)
...
unfreed: <2> test.c(11), 512 bytes at 0x80519e4
{FE FE FE FE FE FE FE FE FE FE FE FE ..............}
```

```
Memory usage statistics (global):
N)umber of allocations made: 2
L)argest memory usage : 1024
T)otal of all alloc() calls: 1024
U)nfreed bytes totals : 512
```

memwatch 显示出有内存错误的程序行。其中，有 double-free 和 unfreed 的 2 个问题。日志结尾部分还有统计信息，包括泄漏了多少内存，使用了多少内存，以及总共分配了多少内存。

更多信息请参考网站：

```
http://www.linkdata.se/sourcecode.html
```

12.4.4 YAMD

YAMD（Yet Another Malloc Debugger）是辅助查找 C 和 C++程序中动态内存分配问题的工具包。

YAMD 具有以下特点。

- 使用处理器的页面管理机制，确定分配内存块的边界。
- 每一步操作都会记录，不仅仅对文件和行号，还有完整的回溯。
- 在底层模拟“malloc”及其相关函数。可以跟踪其他非直接调用“malloc”的函数。
- 不需要修改应用程序源代码。
- 支持通用的内存调试功能。

YAMD 是很有用的内存调试工具。它有一些其他工具没有的特点。但是支持的体系结构还比较少，目前仅支持 X86 平台的 Linux 系统。

YAMD 是由 Nate Eldredge 创建的，是开放源码的软件。从下列站点可以下载 yamd 的源码包。

```
http://www.cs.hmc.edu/~nate/yamd/yamd-0.32.tar.gz
```

解压 yamd-0.32.tar.gz，编译安装工具。

```
# make
# make install
```

缺省地在 YAMD 源码目录下生成 bin/和 lib/目录。分别存放 YAMD 工具和 YAMD 的链接库（包括静态库和动态库）。

YAMD 的源码中的 test 目录下有一些测试程序，是 YAMD 工具使用的例程。不妨分析一下 tests/Makefile。

```
# tests/Makefile
CFLAGS = -g
CC = gcc
# TESTS = 1 2 3 4 5 6 7 8 9
```

```
TEST_BASENAMES = $(basename $(wildcard test*.c))
all : $(addsuffix .exe,$(TEST_BASENAMES))
.PRECIOUS : %.o
%.o : %.c
     $(CC) $(CFLAGS) -c $<
%.dynamic : %.o main.o
     $(CC) -o $@ $^
%.static : %.o main.o ../libyamd.a ../yamd-gcc
     ../yamd-gcc -L.. -o $@ $< main.o
%.exe : %.o main.o ../libyamd.a
     ../yamd-gcc -L.. -o $@ $< main.o
```

通过上面的 Makefile，就可以清楚应用程序的编译有所不同。应用程序编译需要添加"-g"参数，并且需要使用 yamd-gcc 把目标代码和 yamd 库链接成可执行程序。这样，应用程序就包含了 YAMD 的有关目标代码。这样的应用程序不适合在产品中发布。

编译好应用程序，再来测试。以源码中的 test1 为例说明，执行调试命令。

```
# run-yamd test1
```

得到下列结果。

```
YAMD version 0.32
Executable: /usr/src/test/yamd-0.32/test1
...
INFO: Normal allocation of this block
Address 0x40025e00, size 512
...
INFO: Normal allocation of this block
Address 0x40028e00, size 512
...
INFO: Normal deallocation of this block
Address 0x40025e00, size 512
...
ERROR: Multiple freeing At
free of pointer already freed
Address 0x40025e00, size 512
...
WARNING: Memory leak
Address 0x40028e00, size 512
WARNING: Total memory leaks:
1 unfreed allocations totaling 512 bytes
```

```
*** Finished at Tue ... 10:07:15 2002
Allocated a grand total of 1024 bytes 2 allocations
Average of 512 bytes per allocation
Max bytes allocated at one time: 1024
24 K alloced internally / 12 K mapped now / 8 K max
Virtual program size is 1416 K
End.
```

YAMD 显示已经释放了内存，而且存在内存泄漏。

FARSIGHT

第 13 章 部署 Linux 系统

本章目标

本章介绍了 Linux 系统部署的基本方法，分析了文件系统和存储介质的特点。通过本章学习，可以理解文件系统和存储介质之间的关系，并且掌握一些基本的系统部署工具。

- 文件系统类型
- 存储设备类型
- 部署 Linux 系统

13.1 部署 Linux 系统概述

系统部署就是要使目标板的 Linux 系统脱离交叉开发环境，直接在目标机上本地启动运行。由于嵌入式系统硬件的特殊性，特别是存储介质的差异，所以一定要在系统设计阶段就开始考虑。

13.1.1 部署 Linux 系统的基本流程

系统部署仍然要基于嵌入式 Linux 的交叉开发环境。它的前提条件是 Linux 内核和应用程序的开发已经完成，并且在交叉开发环境下测试成功。

通常目标板的存储空间很小，不可能容纳 Linux 主机那样庞大的文件系统，所以定制和裁减文件系统成为首要任务。

定制好了文件系统以后，再配置一个能够挂接本地的文件系统的 Linux 内核，然后把映像都安装到存储设备上去，最后让 Bootloader 引导启动目标板即可。图 13.1 给出了嵌入式 Linux 系统部署流程。

尽管系统部署是最后一项工作，但是这并不是说到产品即将发布的时候再考虑这个问题，在系统设计阶段就要考虑这方面的问题。



图 13.1 Linux 系统部署流程

13.1.2 部署 Linux 系统的关键问题

部署 Linux 系统的关键问题包含 4 个方面。

（1）存储介质

Linux 的文件和数据必须保存在存储介质上。即使内存也是一种存储介质，只不过它不

能永久地保存文件。关键问题是选择什么样的存储介质。影响选择的因素是多方面的，涉及容量、可靠性、性能、价格、应用现场等诸多因素。

网络服务器需要大量数据存储，一般会采用磁盘存储，但是还有 ATA 和 SCSI 不同的类型选择。

手持设备和控制设备需要体积小并且防震等特性，它们就不能使用磁盘存储设备。各种 Flash 可以装配到这类设备上，不过不同的 Flash 的容量、价格、接口、性能等差异也很大。

（2）文件系统

Linux 离不开文件系统，Linux 的程序和文件都是保存在文件系统中的。

Linux 专门为特定的存储设备提供了丰富的文件系统支持。每一种存储介质都有多种文件系统可以选择，但是不同的文件系统的性能各有优缺点。

磁盘存储介质可以选择 EXT2，也可以选择支持日志功能的 EXT3，还有 ReiserFS、JFS、XFS 等文件系统。

Flash 的文件系统也是多样性的，可以选择只读的 CRAMFS，也可以选择可写的 JFFS2，还有 YAFFS 文件系统等。

（3）安装工具

每一种文件系统都有自己的安装、管理和修复工具。这是部署 Linux 系统必需的。通过工具可以把目标板的文件系统安装到存储设备中去。在使用过程中，也有用来修复文件系统的工具。

（4）引导方式

引导程序是 Linux 系统启动所必不可少的。由于存储介质的不同，引导方式有很大差异。从目标板本地的存储设备引导 Linux 系统启动，是 Bootloader 应该具备的功能。

13.2 文件系统类型

文件系统是 Linux 重要的子系统。Linux 采用虚拟文件系统的机制，把所有的东西都看作文件。文件系统是基于块设备驱动程序建立的。目前，Linux 已经能够支持几十种文件系统。

13.2.1 EXT2/EXT3

EXT2（The Second Extended Filesystem）和 EXT3（The Third Extended Filesystem）是 Linux 内核自己的文件系统。

EXT2 发布于 1993 年 1 月，它是由 R\'emy Card、Theodore Ts'o 和 Stephen Tweedie 编写的，它是 EXT 文件系统重写的版本。

EXT2 与传统的 Unix 文件系统有许多共性，都有块（block）、节点（inode）和目录（directory）的概念。尽管没有实现访问控制列表（ACL）、碎片、恢复删除文件和压缩功能，但是都预留了空间。另外，版本兼容机制可以让文件系统添加新的特性（例如：日志），同时保持最大程

度兼容。

在启动的时候，大多数系统要检查文件系统的连续性（执行 e2fsck 命令）。EXT2 文件系统的超级块包含了几个字段，用来表示是否需要执行 fsck。如果文件系统没有卸载干净，或者超出最大挂载数，或者超出最大检查间隔周期，就会执行 fsck。

EXT2 使用的特点兼容机制很复杂。它允许在文件系统中安全地添加许多特点，不会牺牲老版本文件系统代码的兼容性。它的特点兼容机制是从 EXT2 版本 1 引入的，原始的版本 0 并不支持。它有 3 个 32 位字段，一个是兼容的特点（COMPAT），一个是只读兼容的特点（RO_COMPAT），一个是不兼容的特点（INCOMPAT）。

EXT2 元数据操作有异步和同步两种方式。据说异步元数据写操作比 FFS 同步元数据方案快，但是可靠性差一些。这两种方法都可以被相应的 fsck 程序处理。

对于同步写元数据，EXT2 文件系统有 3 种方法。如表 13.1 所示。

表 13.1　　EXT2 文件系统同步元数据

操作对象	操作方法
每个文件（有程序源码）	在 open()函数中使用 O_SYNC 标志
每个文件（没有程序源码）	使用“chattr +S”命令改变文件属性
文件系统	挂接的时候添加“sync”选项（或者在/etc/fstab 中添加）

第 1 种和第 3 种不是 EXT2 文件系统特有的，但是可以强制进行元数据同步写操作。

EXT2 文件系统的磁盘布局会导致各种局限性。当前内核代码的实现也会导致其他一些局限性。许多局限性在文件系统第一次创建的时候就确定了，这取决于块大小的选择。节点与数据块的比例在创建的时候就确定了。因此增大节点数的惟一办法是增大文件系统尺寸，还没有工具能够改变节点和块的比例。

大多数局限性都可以克服，通过磁盘格式的细微调整并且使用兼容标志去适应变化。例如：修改文件系统块大小。

单个目录下最多有 10000～15000 个文件是“软”上限，因为在如此大的目录中创建、删除和查找文件时，线形链表目录实现存在性能问题。使用哈希表的算法可以在单个目录下使用 10 万～100 万以上的文件，而不存在性能问题。单个目录下文件数的绝对上限超过 130 万亿（受文件大小的影响，实际的值要小得多）。

EXT2 的日志功能是 Stephen Tweedie 开发的。日志功能可以避免元数据污损并且需要 e2fsck 检查，而且不需要改变 EXT2 的磁盘布局。总之，日志是用来存储被修改全部元数据块的正规文件，优先于写到文件系统。这意味着可以对已经存在的 EXT2 文件系统创建日志，而不需要数据转换。

当修改文件系统（例如：文件重命名）的时候，事务会存储在日志中，在系统崩溃的时候可以完成或者没有完成事务处理。如果崩溃时事务处理完成，日志的块可以代表有效的文件系统状态，并且复制到文件系统。如果崩溃时事务处理没有完成，就不能保证事务处理的块的连续性（这意味着所代表文件系统修改会丢失）。

EXT3 文件系统是 1999 年 9 月发布的。最早是 Stephen Tweedie 为 2.2 内核版本写的，后来 Peter Braam、Andreas Dilger、Andrew Morton、Alexander Viro、Ted Ts'o 和 Stephen Tweedie

参与移植到 2.4 内核上。

EXT3 是 EXT2 文件系统的改进版，添加了日志等功能。EXT3 使用了全部 EXT2 文件系统的实现，还添加了事务处理的功能。日志功能通过块设备日志层（Journaling Block Device layer，JBD）完成。

JBD 不是 EXT3 文件系统所特有的，它是专门为块设备添加日志功能而设计的。EXT3 文件系统代码会把执行的修改（提交事务）通知 JBD。日志支持事务的启动和停止。在系统崩溃的时候，日志可以快速重新执行事务以保持分区的连续性。事务处理代表对文件系统的单个原子更新操作。JBD 可以在块设备上处理外部日志。

EXT3 的数据模式分为 3 种。

- 写回模式（writeback mode）

对于这种模式的数据，EXT3 根本不做日志；在 XFS、JFS 和 ReiserFS 文件系统中，它缺省地提供了简单的元数据日志。崩溃重启可能引起正在写的数据出错。在这种模式下，EXT3 文件系统性能最好。

- 有序模式（ordered mode）

对于这种模式的数据，EXT3 仅正式地做元数据日志，但是逻辑上把元数据和数据块组成一个事务单元。在向磁盘上写元数据之前，先写相关的数据块。这种模式性能比写回模式略微慢一点，但是比下面的日志模式快很多。

- 日志模式（journal mode）

对于这种模式的数据，EXT3 将对全部数据和元数据做日志处理。所有新的数据先写到日志区，然后写到它最终的位置。遇到崩溃事件，日志可以重做，保持数据和元数据的连续性。这种模式是最慢的，除了数据需要做同时从磁盘读出并且写回操作的情况。

EXT3 文件系统完全兼容 EXT2，EXT2 分区可以挂接成 EXT2 格式。EXT2 分区可以通过 tune2fs 命令转换成 EXT3 格式。文件系统工具见表 13.2 的说明。

表 13.2 EXT2 文件系统工具

工 具 名 称	说 明
tune2fs	通过-j 标志在 EXT2 分区上创建 EXT3 的日志
mke2fs	创建 EXT2 文件系统，通过-j 标志可以创建 EXT3 文件系统
debugfs	ext2 和 ext3 文件系统调试工具

有关 EXT2 文件系统工具程序（e2fsck）可以参考网站：

```
http://e2fsprogs.sourceforge.net/
```

EXT3 文件系统的使用可以参考：

```
http://www.zip.com.au/~akpm/linux/ext3/ext3-usage.html
```

13.2.2 JFS

JFS（Journaled File System，日志文件系统）是 IBM 创建的一种文件系统。

JFS 提供了基于日志的字节级文件系统，它是为面向事务的高性能系统而开发的。它具

有可伸缩性和健壮性，与非日志文件系统相比，具有快速重启的优点。与 EXT3 不同，JFS 采用完全内部集成的日志功能，而不是在已经存在的文件系统上添加日志。

从设计角度来说，JFS 具有以下特性。

（1）日志处理

JFS 使用原来为数据库开发的技术，记录了文件系统元数据上执行的操作（即原子事务）信息。如果发生系统故障，可通过重放日志并对适当的事务应用日志记录，来使文件系统恢复到一致状态。由于重放实用程序只需检查文件系统最近活动所产生的运行记录，而不是检查所有文件系统的元数据，因此，与传统的文件系统相比，这种基于日志的方法相关的文件系统恢复时间较快。

基于日志恢复的其他几个方面也值得注意。首先，JFS 只记录元数据上的操作，因此，重放这些日志只能恢复文件系统中结构关系和资源分配状态的一致性。它没有记录文件数据，也没有将这些数据恢复到一致状态。因此，恢复后某些文件数据可能丢失或失效，对数据一致性有关键性需求的用户应该使用同步 I/O。

面对存储介质出错，日志记录不是特别有效。特别地，在将日志或元数据写入磁盘的期间发生的 I/O 错误，意味着在系统崩溃后，要将文件系统恢复到一致状态，需要耗时并且有可能强加的全面完整性检查。这暗示着，坏块重定位是任何驻留在 JFS 下的存储管理器或设备的一个关键特性。

（2）基于盘区的寻址结构

JFS 使用基于盘区的寻址结构，连同主动的块分配策略，产生紧凑、高效、可伸缩的结构，以将文件中的逻辑偏移量映射成磁盘上的物理地址。盘区是像一个单元那样分配给文件的相连块序列，可用一个由<逻辑偏移量，长度，物理地址>组成的三元组来描述。寻址结构是一棵 B+树，该树由盘区描述符（上面提到的三元组）填充，根在 inode 中，键为文件中的逻辑偏移量。

（3）可变的块尺寸

按文件系统分，JFS 支持 512、1024、2048 和 4096B 的块尺寸，以允许用户根据应用环境优化空间利用率。较小的块尺寸减少了文件和目录中内部存储碎片的数量，空间利用率更高。但是，小块可能会增加路径长度，与使用大的块尺寸相比，小块的块分配活动可能更频繁发生。因为服务器系统通常主要考虑的是性能，而不是空间利用率，所以缺省块尺寸为 4096B。

（4）动态磁盘 inode 分配

JFS 按需为磁盘 inode 动态地分配空间，同时释放不再需要的空间。这一支持避开了在文件系统创建期间，为磁盘 inode 保留固定数量空间的传统方法，因此用户不再需要估计文件系统包含的文件和目录最大数目。另外，这一支持使磁盘 inode 与固定磁盘位置分离。

（5）目录组织

JFS 提供 2 种不同的目录组织。第 1 种组织用于小目录，并且在目录的 inode 内存储目录内容。这就不再需要不同的目录块 I/O，同时也不再需要分配不同的存储器。最多可有 8 个项可直接存储在 inode 中，这些项不包括自己(.)和父(..)目录项，这 2 个项存储在 inode 中不

同的区域内。

第 2 种组织用于较大的目录，用按名字键控的 B+树表示每个目录。与传统无序的目录组织比较，它提供更快的目录查找、插入和删除能力。

（6）稀疏和密集文件

按文件系统分，JFS 既支持稀疏文件也支持密集文件。

稀疏文件允许把数据写到一个文件的任意位置，而不要将以前未写的中间文件块实例化。所报告的文件大小是已经写入的最高块位处，但是，在文件中任何给定块的实际分配，只有在该块进行写操作时才发生。例如，假设在一个指定为稀疏文件的文件系统中创建一个新文件。应用程序将数据块写到文件中第 100 块。尽管磁盘空间只分配了 1 块给它，JFS 将报告该文件的大小为 100 块。如果应用程序下一步读取文件的第 50 块，JFS 将返回填充了 0 的一个字节块。假设应用程序然后将一块数据写到该文件的第 50 块，JFS 仍然报告文件的大小为 100 块，而现在已经为它分配了 2 块磁盘空间。稀疏文件适合需要大的逻辑空间但只使用这个空间的一个（少量）子集的应用程序。

对于密集文件，将分配相当于文件大小的磁盘资源。在上例中，第一个写操作（将一块数据写到文件的第 100 块）将导致把 100 个块的磁盘空间分配给该文件。在任何已经隐式写入的块上进行读操作，JFS 将返回填充了 0 的字节块，正如稀疏文件的情况一样。

（7）文件系统大小和文件长度

JFS 支持的最小文件系统是 16MB。最大文件系统的大小是文件系统块尺寸和文件系统元数据结构支持的最大块数两者的乘积。JFS 将支持最大文件长度是 512 万亿字节（TB）（块尺寸是 512B）到 4 千万亿字节（PB）（块尺寸是 4KB）。

最大文件长度是主机支持的虚拟文件系统的最大文件长度。例如：如果主机只支持 32 位，则这就限制了文件长度。

JFS 文件系统已经被 Linux 2.6 内核采纳。挂接 JFS 文件系统的选项如表 13.3 所示。

表 13.3　　JFS 文件系统挂接选项

挂 接 选 项	含　　义
iocharset=name	可以把 Unicode 字符集转换到 ASCII 字符集。缺省的是不作转换。使用 iocharset=utf8，转换成 UTF8 字符集。这时还要在内核中配置 CONFIG_NLS_UTF8 选项
resize=value	改变 volume 的块数。JFS 只支持增大 volume，不能减小 volume。这个选项只在 remount 的时候有效
nointegrity	不做写日志工作。这个选项的基本用法是在从备份元数据中回复 volume 的时候，力求最高性能。如果系统非正常的停止，这个 volume 的完整性不能保证
integrity	这是缺省值。保存元数据到日志区
errors=continue	出错时继续执行
errors=remount-ro	这是缺省值。出错时重新挂接成只读的
errors=panic	出错时系统 panic 并且停止运行

有关 JFS 的开发可以参考下面的网站。

```
http://jfs.sourceforge.net/
```

13.2.3 cramfs

cramfs 是专门为小而且简单的文件系统设计的，用于在 ROM 芯片或者 CD 上存储文件系统。

它压缩比很高。它使用 zlib 函数，一次压缩文件的一个页，并且允许随机页访问。元数据（meta-data）不会被压缩，但是用非常简单的表达方式使它比传统文件系统使用的磁盘空间更小。

cramfs 文件系统具有以下特点。

- cramfs 文件系统不能支持写操作（文件系统是压缩的，很难瞬时修改文件），因此需要使用“mkcramfs”工具制作磁盘映像。
- 文件大小限制在 16MB 以内。
- 最大的文件系统尺寸略大于 256MB。在文件系统中的最后一个文件允许超出 256MB 的限制。
- 只保存 GID 的低 8 位。cramfs 当前的版本仅截取 8 位，这存在潜在的安全问题。
- cramfs 映像支持硬连接，但是被连接文件的连接数只能是 1。
- cramfs 文件系统没有“.”和“..”条目。目录总是有连接数 1。（使用 find 命令的选项“-noleaf”是没有用的）
- 在 cramfs 中不保存时间戳，因此缺省的时间都是起始值（1970 年）。最近访问的文件可以更新时间戳，但是仅当 inode 缓存在内存中的时候有效，这个时间戳不能保存下来。

目前，cramfs 必须以与处理器体系结构相同的端（Endian）读写，只能在内核中以 PAGE_CACHE_SIZE 等于 4096 读取。如果有更大的页，可以调整 mkcramfs.c 中的宏定义，只要不怕这个文件系统不能被其他内核读取就行。

cramfs 映像中包含固定的格式信息，下列数据的说明了 cramfs 存储格式。其中，第 0 和第 512 个字节是 cramfs 的识别码“0x28cd3d45”，紧接着是存储描述。

```
0   ulelong     0x28cd3d45  Linux cramfs offset 0
>4  ulelong     x       size %d
>8  ulelong     x       flags 0x%x
>12 ulelong     x       future 0x%x
>16 string  >\0    signature "%.16s"
>32 ulelong     x       fsid.crc 0x%x
>36 ulelong     x       fsid.edition %d
>40 ulelong     x       fsid.blocks %d
>44 ulelong     x       fsid.files %d
>48 string  >\0    name "%.16s"
512 ulelong 0x28cd3d45      Linux cramfs offset 512
>516 ulelong    x       size %d
>520 ulelong    x       flags 0x%x
```

```
>524 ulelong    x        future 0x%x
>528 string >\0    signature "%.16s"
>544 ulelong    x        fsid.crc 0x%x
>548 ulelong    x        fsid.edition %d
>552 ulelong    x        fsid.blocks %d
>556 ulelong    x        fsid.files %d
>560 string >\0    name "%.16s"
```

因为 cramfs 是只读的文件系统，所以它的内容必须在创建的时候就确定好。生成映像以后，可以烧写到 Flash/ROM 芯片上，由 Linux 内核挂接。

通常 cramfs 可以结合其他文件系统使用，并且可以基于 MTD 设备使用。

13.2.4 JFFS/JFFS2

JFFS（Journaling Flash Filesystem）是瑞典的 Axis 通讯公司（Axis Communications AB）设计开发的。

JFFS2（Journaling Flash Filesystem Version 2）是 RedHat 公司基于 JFFS 文件系统开发的，它是 JFFS 的改进版。

JFFS 和 JFFS2 都是开源的日志文件系统，最适合在 Flash 芯片上使用。它们的日志结构能够保持文件系统的连续性。即使文件系统崩溃或者非正常掉电，重启的时候也不需要执行 fsck。另外，它们还考虑了 Flash 存储介质的物理特点。

JFFS 是完全日志结构的。这个文件系统就相当于 Flash 介质上的大量节点列表。每一个节点（jffs_node 结构体）包含了有关文件的一些信息，也可以包含这个文件名，还有一些数据。在数据存在的情况下，jffs_node 会包含一个字段，用来说明那些数据在文件中的位置。这样，新数据可以覆盖老数据。

除了普通的 inode 信息，jffs_node 还包含了一个字段，用来说明在节点给定偏移地址删除多少数据。这用于截取文件等操作。

每个节点也还有一个版本"version"号，从写到文件的第一个节点开始为"1"，以后每写一个新节点就加"1"。这些节点的顺序无关紧要，但是为了保持擦除均匀，总是从头开始写，一直写到结尾才执行擦除操作。

为了重建文件内容，可以扫描整个介质（参考在挂接时调用的 jffs_scan_flash()函数），并且把单个节点放入递增的"version"序列。在每一个应该插入/删除数据的地方解释指令。当前文件名就是那个包含名字字段的最新节点。

在整个节点列表到达介质末尾之前，这样处理很简单。之后，就必须从头开始了。在第一个擦除块的节点中，有些可能已经被后面的节点废弃。因此，在实际到达 Flash 结尾之前，完全地填充文件系统，从仍然有效的第一个块复制所有的节点，并且擦除原始块。希望这样可以给我们更多空间。如果没有，继续处理下一个块等。这叫作垃圾回收。

注意必须确保永远不要出现的一种状态：头部正在写新节点，尾部是最老的节点，这时两者之间的空闲区域都用光了。这意味着根本不能继续进行垃圾回收，即使有些废弃的节点，文件系统也可能阻塞。

尽管现在是从头开始使用到末尾，但是它应该分别处理擦除块，并且用几种状态（free/filling/full/obsoleted/erasing/ bad）保存擦除块列表。总之，块会在 free->erasing 列表中继续，然后返回到 free（通过重写任何仍然有效的节点到“filling”节点）。

有关 JFFS 的信息参考网站：

```
http://developer.axis.com/software/jffs/
```

JFFS2 是 JFFS 的改进版。它在下列方面有些改进。

（1）了解和处理按照擦除扇区（Sector）级写 Flash。这样做有各种好处，例如垃圾回收可以基于扇区而不是整个文件系统。

（2）能够标记坏块扇区并且继续使用剩余的好扇区，这样可以提高设备使用寿命。

（3）垃圾回收导致的阻塞时间更少。最小可以只擦除一个扇区，不像 JFFS 需要把整个文件系统数据都压到垃圾回收区。

（4）文件系统设计就提供了本地数据压缩。

JFFS2 设计支持 ROM、NOR Flash 和 NAND Flash 芯片。支持磨损平衡，从而延长 Flash 寿命。运行时总是把 Flash 目录结构保存在 RAM 中，提高系统性能。采用压缩的格式存储数据可以存储更多文件。

有关 JFFS2 的信息参考网站：

```
http://www.linux-mtd.infradead.org/
```

13.2.5 YAFFS

YAFFS（Yet Another Flash Filing System）是 Charles Manning 为 Aleph One 公司设计开发的，它是第一种专门为 NAND Flash 设计的文件系统。

YAFFS 基于日志的文件系统，提供磨损平衡和掉电恢复的鲁棒性。它还为大容量的 Flash 芯片做了很好的调整，针对启动时间和 RAM 的使用做了优化。它适用于大容量的存储设备，已经在 Linux 和 WinCE 商业产品中使用。

YAFFS 充分考虑了 NAND 闪存的特点，根据 NAND 闪存以页面为单位存取的特点，将文件组织成固定大小的数据段。利用 NAND 闪存提供的每个页面 16B 的备用空间来存放 ECC（Error Correction Code）和文件系统的组织信息，不仅能够实现错误检测和坏块处理，也能够提高文件系统的加载速度。YAFFS 采用一种多策略混合的垃圾回收算法，结合了贪心策略的高效性和随机选择的平均性，达到了兼顾损耗平均和系统开销的目的。

YAFFS 将文件组织成固定大小（512B）的数据段。每个文件都有一个页面专门存放文件头，文件头保存了文件的模式、所有者 id、组 id、长度、文件名等信息。为了提高文件数据块的查找速度，文件的数据段被组织成树形结构。YAFFS 在文件进行改写时总是先写入新的数据块，然后将旧的数据块从文件中删除。YAFFS 使用存放在页面备用空间中的 ECC 进行错误检测，出现错误后会进行一定次数的重试，多次重试失败后，该页面就被停止使用。

YAFFS 充分利用了 NAND 闪存提供的每个页面 16B 的备用空间，参考了 SmartMedia 的方案，备用空间中 6 个字节被用作页面数据的 ECC，2 个字节分别用作块状态字和数据状态

字，其余的 8 字节（64 位）用来存放文件系统的组织信息。由于文件系统的基本组织信息保存在页面的备份空间中，因此，在文件系统加载时只需要扫描各个页面的备份空间，即可建立起整个文件系统的结构，而不需要像 JFFS 那样扫描整个介质，从而大大加快了文件系统的加载速度。

YAFFS 中用数据结构来描述每个擦除块的状态。该数据结构记录了块状态，并用一个 32 位的位图表示块内各个页面的使用情况。在 YAFFS 中，有且仅有一个块处于“当前分配”状态。新页面从当前进行分配的块中顺序进行分配，若当前块已满，则顺序寻找下一个空闲块。

YAFFS 使用一种多策略混合的算法来进行垃圾回收，将贪心策略和随机选择策略按一定比例混合使用：当满足特定的小概率条件时，垃圾回收器会试图随机选择一个可回收的页面；而在其他情况下，则使用贪心策略回收最“脏”的块。通过使用多策略混合的方法，YAFFS 能够有效地改善贪心策略造成的不平均；通过不同的混合比例，则可以控制损耗平均和系统开销之间的平衡。考虑到 NAND 的擦除很快（和 NOR 相比可忽略不计），YAFFS 将垃圾收集的检查放在写入新页面时进行，而不是采用 JFFS 那样的后台线程方式，从而简化了设计。

YAFFS 的核心是 YAFFS/direct，可以方便地合并到实时操作系统和嵌入式操作系统中。可以获取到引导程序和文档。尽管设计目的是为了保留 NAND Flash 的使用效率，但是它也能支持 NOR Flash 和 RAM。

YAFFS2 是 YAFFS 的第 2 个版本。YAFFS 版本 1 支持具有 512B 页和 16B 备用空间（OOB）的 NAND Flash，但是不能支持具有 2048 字节页和 64 字节备用空间的新 Flash。YAFFS2 更适合这些新的芯片，它支持的页面更大，性能更好。

YAFFS/direct 代码可以基于 GPL 或者产品专利获取。

目前 Linux 内核还没有正式支持 YAFFS，所以需要通过补丁修改 Linux 内核。另外，YAFFS 也需要 MTD 设备驱动的支持。

更多 YAFFS 的信息参考以下网站。

```
http://www.aleph1.co.uk/armlinux/projects/yaffs/index.html
```

13.3 存储设备

嵌入式系统的引导程序和 Linux 映像都需要永久保存。根据不同嵌入式应用的需求，可以选择不同的存储设备。在使用之前，首先需要了解 Linux 对这些存储设备的支持程度。

13.3.1 MTD 类型设备

MTD（Memory Technology Device）是 Linux 内核采纳的一种设备子系统，它为底层的存储芯片提供了统一的设备接口。MTD 子系统接口如图 13.2 所示。

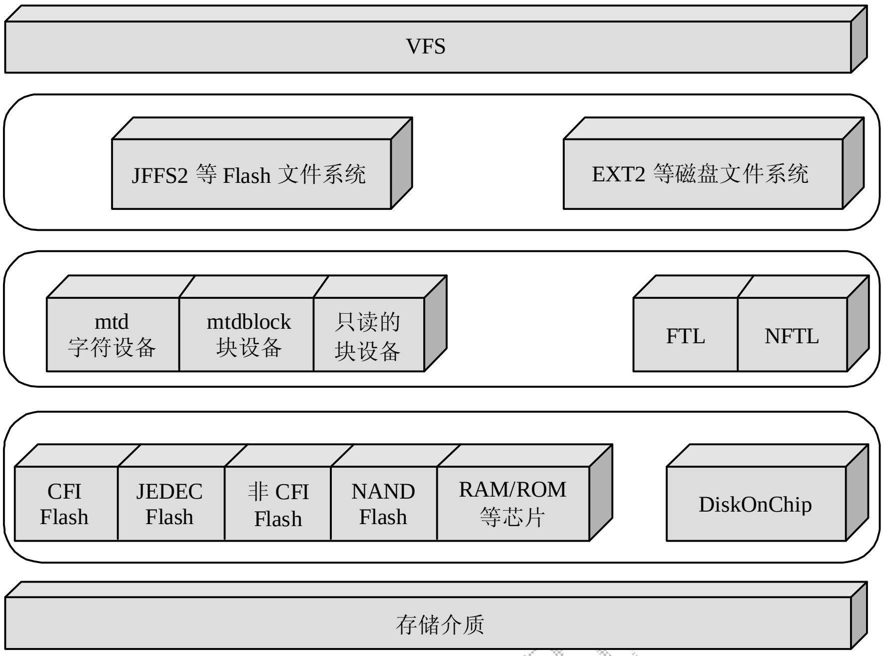


图 13.2　MTD 子系统接口图

MTD 芯片驱动程序必须向 MTD 子系统注册，通过结构体 mtd_info 给 add_device()函数提供一组缺省的回调函数和属性。MTD 驱动必须实现这些回调函数，让 MTD 子系统能够通过函数调用执行删除、读出、写入和同步等操作。

MTD 子系统同时可以提供 2 类 MTD 驱动程序。一类驱动程序是 MTD 设备地址空间的映射，提供直接访问设备的操作，属于字符设备驱动；另一类驱动程序则为建立文件系统提供基础，属于块设备驱动。

内核配置界面 MTD 子菜单的选项如下。

（1）“Direct char device access to MTD devices”

这是直接访问的字符设备驱动程序，它把每一个 MTD 设备对应成一个字符设备，允许用户直接读写存储芯片，并且可以使用 ioctl()函数去获取设备的信息，或者擦除部分。

（2）“Caching block device access to MTD devices”

这是缓存方式访问的块设备驱动程序，它把每一个 MTD 设备对应成一个块设备，支持 Flash 的块擦除等操作。它是建立 Flash 文件系统的基础。通常需要在它上面安装 JFFS/JFFS2 文件系统，然后再挂接 mtdblock 设备。

（3）“Readonly block device access to MTD devices”

这是只读方式访问的块设备驱动程序。它可以从 MTD 设备上挂接只读的文件系统（例如：cramfs），免除了驱动程序数据缓冲的花销。

（4）“FTL（Flash Translation Layer）support”

这是 FTL（Flash 转换层）驱动程序。它为 PCMCIA 标准的原始 Flash 转换层提供支持。它在 Flash 设备上使用一种伪文件系统来仿真块设备的 512B 扇区，从而在设备上建立普通的文件系统。

这段代码的算法是受专利保护的。尽管它可以按照 GPL 去复制、修改、发布这些代码，但是它在美国只允许用到 PCMCIA 的硬件设备上。

（5）“NFTL（NAND Flash Translation Layer）support”

这是 NFTL（NAND Flash 转换层）驱动程序。它为 M-Systems 的 DiskOnChip 设备的 NANDFlash 转换层提供支持。它在 Flash 设备上使用一种伪文件系统来仿真块设备的 512B 扇区，从而在设备上建立普通的文件系统。

这段代码的算法也是受专利保护的。尽管它可以按照 GPL 去复制、修改、发布这些代码，但是它在美国只允许用到 DiskOnChip 硬件设备上。

MTD 能够支持各种 ROM、RAM、Flash（NOR 和 NAND）以及 DOC（M-Systems 的 DiskOnChip）等存储芯片。因为各种芯片的特点和功能不同，它们也需要专有的工具和操作方法。

13.3.2　磁盘类型设备

（1）ATA/ATAPI

ATA（AT Attachment）的名字来源于 IBM PC/AT（1984）计算机，它是计算机内部磁盘驱动器使用的并行接口。它习惯上叫作 IDE（Integrated Drive Electronics）接口。因为它的 40 针电缆符合 IBM PC AT 的 ISA 系统总线限制，所以真正的名字应该是 ATA。

ATAPI（AT Attachment Packet Interface）是扩展的 ATA 接口，它支持 CD/DVD、磁带机和一些特殊移动存储设备（ZIP 和 LS-120）等设备。ATAPI 驱动通过 SCSI 命令包控制 ATAPI 设备。SCSI 命令包通过 ATA 接口传输，而不是通过 SCSI 总线。

ATA/ATAPI 接口最早用于支持 PC AT 计算机的硬盘。它是目前计算机流行的的磁盘接口，可以支持 IDE 硬盘、CD/DVD 光驱等设备。

ATA/ATAPI 接口标准的主要维护组织有 ANSI T10 和 T13。当前的 ATA/ATPAI 已经有了许多版本，例如：SATA（串行 ATA）已经在新的 PC 上得到广泛支持；SATAPI（串行 ATAPI）可以通过串行 ATAPI 支持 CD/DVD 和磁带机等设备；还有 CF（Compact Flash）也可以基于 ATA 的接口，有相关的版本支持。

（2）SCSI

SCSI（Small Computer System Interface）是一种并行接口标准，在苹果公司的 Macintosh 计算机、PC 和许多 Unix 系统上作为外围设备接口。

SCSI 接口提供了比标准串口和并口更快的数据传输速率（达到 80MB/s）。另外，还可以在一个 SCSI 口上连接许多设备，因此，SCSI 是真正的 I/O 总线。

尽管 SCSI 是一种 ANSI 标准，但是它有许多变体，而且 2 种 SCSI 接口可以是不兼容的。例如：SCSI 支持几种类型的连接器。

SCSI 是一个通用接口，可以连接各种外围硬件。通常它可以用作连接硬盘的高性能接口。SCSI 存储设备在嵌入式系统上应用比较少见，但是在高可靠性的网络存储设备和服务器上常用。

SCSI 接口已经是一个用户群相当大的成熟技术，Linux 对 SCSI 的驱动程序也非常完善。这种标准的驱动程序接口可以用于其他存储设备的接口。例如：USB 存储设备的驱动程序就是仿真成 SCSI 存储设备使用，因此它也需要 SCSI 接口的驱动程序。

（3）其他存储设备

随着计算机系统结构的发展，存储设备的接口类型越来越多。除了 ATA 和 SCSI 的接口的硬盘设备，还有 MMC（Multi-Media Card）存储卡、SMC（Smart Media Card）存储卡、USB 存储盘等，在数字消费设备上应用极为广泛。

下面是有关的组织或者链接。

ANSI（American National Standards Institute）

```
http://www.ansi.org
```

INCITS（InterNational Commitee for Information Technology Standards）

```
http://www.incits.org/
```

T13（Technical Committee 13，ATA/ATAPI 委员会）

```
http://www.t13.org/
```

T10（Technical Committee 10，SCSI 委员会）

```
http://www.t10.org/
```

SATA-I/O（Serial ATA’s secret society）

```
http://www.sata-io.org/
```

PCMCIA（Personal Computer Memory Card International Association）

```
http://www.pcmcia.org/
```

CF 协会（Compact Flash Association）

```
http://www.compactflash.org/
```

MMC 协会（Multi Media Card Association）和 SD 协会（SD Card Association）

```
http://www.mmca.org/
```

CE-ATA

```
http://www.ce-ata.org/
```

13.4 部署 Linux 系统

不同的存储设备需要不同的方法和工具来部署 Linux 系统。

13.4.1 安装 MTD 工具

在 MTD 设备上部署文件系统的时候，需要一套 MTD 工具，可以擦除或者格式化 MTD

设备。这些工具都包含在 MTD 源码包中，但是针对不同的内核版本，需要选择适当的 MTD 版本。

从下列站点可以下载到 CVS 快照，这里提供最新的源码包，包含全部 MTD 源代码。

```
ftp://ftp.uk.linux.org/pub/people/dwmw2/mtd/cvs/mtd-snapshot-20060403.tar.bz2
```

这里的 mtd-snapshot-20060403.tar.bz2 是源码包的名字。解压完成后，就可以进行配置编译 mtd 工具了。

有些 MTD 工具必须安装到目标机上执行，例如：flash_erase。也有些既可以在开发主机上使用，也可以在目标机上使用，例如：mkfs.jffs2。这样就需要分别为开发主机和目标机安装编译这些工具。

（1）为开发主机安装 MTD 工具

```
cd /mtd/util
automake --foreign; autoconf
./configure --with-kernel=/usr/src/linux
make clean
make
make prefix=/usr install
```

这样工具已经安装到/usr/sbin 目录下了，其中包含 mkfs.jffs2 工具。

如果在主机端可以使用移动 MTD 存储设备，就需要创建 MTD 设备的节点。在 mtd/utils 目录下有 MAKEDEV 的脚本，到/dev 目录下执行这个脚本，就可以自动创建 MTD 相关的节点。

（2）为目标机安装 MTD 工具

大多数嵌入式系统使用板上 Flash，无法移到主机端操作，因此还需要在目标机文件系统中安装 MTD 工具。

目标机安装使用 MTD 工具的时候，需要 zlib 库的支持。Zlib 库可以从 http://www.gzip.org 下载。按照下列步骤编译安装共享的 zlib 库。

```
cd zlib-1.1.4
CC=arm-linux-gcc LDSHARED="arm-linux-ld -shared" ./configure --shared
make
make prefix=/usr/local/arm/3.3.2/rootfs install
```

这样 zlib 库就安装到/usr/local/arm/3.3.2/rootfs/lib 目录下了。

接下来为目标机安装 MTD 工具集。修改 mtd/util/Makefile 中的 CROSS 变量定义为交叉编译器的前缀。

```
CROSS = arm-linux-
```

然后交叉编译安装 MTD 工具。

```
cd mtd/util
automake --foreign; autoconf
./configure --with-kernel=~/linux-2.6.14
make clean
make
make prefix=/usr/local/arm/3.3.2/rootfs install
```

这样，目标机的 MTD 工具就安装到/usr/local/arm/3.3.2/rootfs/sbin 目录下了。后面部署 MTD 的操作主要使用目标机端的工具。

13.4.2 使用磁盘文件系统

最常用的磁盘存储设备应该是 IDE 硬盘，几乎所有的 PC 计算机上都使用 IDE 硬盘。因为它具有存储容量大、价格低的优势，有些需要大存储容量的嵌入式设备也会采用。

在 Linux 系统上，IDE 硬盘设备对应的设备节点是/dev/hda /dev/hdb /dev/hdc /dev/hdd。分别对应 2 个 IDE 口上的 4 块硬盘（每个 IDE 口最多挂接主从两块硬盘）。在交叉开发环境下，硬盘驱动程序已经加载，通过/dev/hda 等设备节点可以访问。

如同开发主机安装 Linux 系统一样，首先要对硬盘分区。这可以通过 Linux 系统工具 fdisk 完成。具体 fdisk 命令的使用可以通过 man 命令查看手册。习惯上，硬盘分区和设备节点对应如表 13.4 所示，更多分区依此类推。

表 13.4 硬盘分区与设备节点关系

设 备 节 点	对应硬盘分区
/dev/hda	整块硬盘
/dev/hda1	硬盘第一个主分区
/dev/hda2	硬盘第二个主分区
/dev/hda4	硬盘扩展分区
/dev/hda5	硬盘第一个逻辑分区
/dev/hda6	硬盘第二个逻辑分区

为硬盘划分好分区，接下来就要制作文件系统了。Linux 的制作文件系统的概念等同于 Windows 的格式化磁盘操作。这需要文件系统制作工具，而不同类型的文件系统分别有各自的工具。每一种文件系统的制作工具的命名格式为 mkfs 加上文件系统类型的扩展名。例如：EXT2 的工具是 mkfs.ext2；EXT3 的工具是 mkfs.ext3；JFS 的工具是 mkfs.JFS 等等。

然后挂接文件系统，就可以在硬盘分区上存取文件了。把为目标板定制的文件系统全部复制到这个分区下。一个不用“cp”命令的操作可以完成这项功能，分析下面的操作过程。

以 EXT3 文件系统为例，把～/myfs 目录下的文件全部复制到/dev/hda1 分区上。

```
# mkfs.ext3  /dev/hda1
# mount  /dev/hda1  /mnt
```

```
# (cd ~/myfs && tar cf - .) | (cd /mnt && tar xvf -)
# umount /mnt
```

磁盘文件系统还有一个检查、修复工具 fsck。这个命令可以检查文件系统的连续性，修复发现的问题。

当系统突然掉电或者崩溃以后，可能导致文件系统不连续。对于 EXT2 文件系统，一般需要 fsck 才能恢复文件系统连续性，而检查文件系统又需要很长时间。对于日志文件系统，文件系统会根据日志恢复文件系统。如果文件系统不认为是干净的，这意味着由于某种原因没有完整和正确地重放日志，或者文件系统不能单靠重放日志来恢复到一致状态，那么，就对文件系统执行一遍完整检查。

13.4.3 使用 RAMDISK 设备

RAMDISK 就是把指定的内存区域模拟成磁盘设备，它属于块设备驱动程序。

基于 RAMDISK 的块设备，可以建立 EXT2 格式的磁盘文件系统。在内核启动之前，通常需要把 EXT2 文件系统的压缩镜像解压到内存指定位置，然后就可以把 RAMDISK 设备挂载成根文件系统。

RAMDISK 的最大特点是运行速度快，因为文件系统内容全部保存在内存中。反过来就成了缺点，因为它会占用一些物理内存，而且系统重启无法保存上次运行中的信息。所以，RAMDISK 比较适合较小并且不需要永久保存数据的文件系统。

在磁盘文件系统中，/boot/initrd-x.x.x.img 文件就是一个 RAMDISK 映像。Initrd（Initial RAMDISK）一般可以用来辅助引导 Linux 系统。它包含一个基本的文件系统和必要的驱动程序（例如：EXT3）以及文件系统检查修复工具。还包含一个 linuxrc 的脚本，执行 initrd 具体操作命令，实现加载一些模块和安装文件系统等。

启动过程中，Bootloader 先把 initrd 映像加载到内存，内核再把 RAMDISK 设备挂接成根文件系统，然后加载需要的驱动程序并且挂接真正的硬盘分区，最后把硬盘分区的文件系统转换成根文件系统。

在使用 Flash 的嵌入式系统中，RAMDISK 一般可以直接作为根文件系统使用，也可以结合可存储的文件系统使用，例如：把一块 JFFS2 类型的文件系统 MTD 分区挂接到指定目录下。

在 Linux 主机系统下面，通常可以通过下列步骤制作 RAMDISK 映像。

（1）创建空的文件系统映像。

```
$ dd if=/dev/zero of=initrd.img bs=1k count=4096
$ mkfs.ext2 -F initrd.img
```

上面的命令创建了一个 4096KB 的文件系统镜像，并通过/dev/zero 设备进行初始化清空。

（2）作为 loop 设备挂接 RAMDISK 映像。

```
$ mkdir /mnt/initrd
$ mount –t ext2 -o loop initrd.img /mnt/initrd
```

这样，/mnt/initrd 目录就对应 initrd.img 存储设备了。

（3）创建目录并安装文件

```
$ cd /mnt/initrd
$ mkdir bin dev etc lib mnt proc sbin sys usr
```

然后再创建设备节点，添加相应的程序。在第 10 章已经为制作文件系统做了描述。如果已经定制好了一个文件系统，全部复制过来即可。

（4）压缩映像

把目标板需要的文件系统内容添加上去以后，先要把 loop 设备卸载下来，然后用 gzip 命令把映像压缩一下。

$ cd ～/

$ umount /mnt/initrd

$ gzip --best -c initrd.img > initrd.img.gz

这样一个压缩的 RAMDISK 映像就制作好了。使用 RAMDISK 还需要配置内核命令行参数，指定 RAMDISK 的物理内存地址。假如把上面 4MB 的 RAMDISK 解压到 0x30800000，内核的命令行参数应该包含下列内容。

```
root=/dev/ram rw initrd=0x30800000,4M
```

另外，还可以使用转换成 U-Boot 的格式，这样 U-Boot 可以识别这个 RAMDISK 映像，并且解压到内存中。

```
$ mkimage -n 'RAM disk' -A arm -O linux -T ramdisk -C gzip -d initrd.img.gz
initrd.uboot
```

13.4.4 使用 MTD 设备和 JFFS2 文件系统

Linux 内核的 MTD 驱动可以支持分区功能，它可以把一块 Flash 分成几个区。比如可以分成 Boot、Kernel 和 Filesystem 分区，分别存储 Bootloader、内核和文件系统。

对于 Flash 的分区表是通过 mtd_partition 结构体来描述的。对于不同的目标板，既可以通过引导程序传递分区参数，也可以直接在程序中定义。以内核源码的 driver/mtd/maps/physmap.c 为例，它可以支持从内核命令行参数或者 Redboot 读取分区表，也可以直接添加下列程序定义结构体。

```
static struct mtd_partition physmap_partitions[] = {
    {
        .name =     "Bootloader",
        .size =     0x00040000,
        .offset =   0,
        .mask_flags =   MTD_WRITEABLE  /* force read-only */
    },{
```

```
            .name =     "Kernel",
            .size =     0x00100000,
            .offset =   0x00040000,
        },{
            .name =     "Filesystem",
            .size =     MTDPART_SIZ_FULL,
            .offset =   0x00140000
        }
    };
```

为 MTD 驱动添加分区表并且驱动成功以后，在目标机端可以看到 MTD 有关的其他启动信息。还可以通过/proc/mtd 接口查看分区信息。

```
# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00040000 00020000 "Bootloader"
mtd1: 00100000 00020000 "Kernel"
mtd2: 0ec00000 00020000 "Filesystem"
```

接下来制作 JFFS2 的文件系统。通过 mkfs.jffs2 工具可以把为目标板定制好的文件系统目录转换成一个 JFFS2 映像。假设定制文件系统目录 rootfs 在当前目录下，执行下列命令。

```
# mkfs.jffs2  -r rootfs -o rootfs.jffs2
```

然后通过 MTD 工具把内核映像和文件系统映像写到对应的 MTD 分区。对于板上的 Flash，MTD 工具需要运行在目标板上。

```
# flash_eraseall /dev/mtd1
# cp uImage /dev/mtd1
# flash_eraseall /dev/mtd2
# cp rootfs.jffs2 /dev/mtd2
```

这样就可以把内核映像和文件系统映像写到 MTD 中了。可以再检查一下 JFFS2 文件系统是否能够正常挂接。

```
# mount -t jffs2 /dev/mtd/block2 /mnt
# ls /mnt
# umount /mnt
```

最后要让目标板挂接本地的文件系统，还要修改命令行参数 root。

```
root=/dev/mtdblock2 rw
```

13.4.5 系统启动和升级

任何一个软件产品都有可能修改和升级，那么系统必须具备适当的途径更新文件内容或

者映像。

对于磁盘文件系统来说，存储容量不是大问题，可以远程下载文件，实现打补丁或者替换。

对于 Flash 存储设备来说，内核和文件系统一般都是作为映像烧写到 Flash 中的。因此升级的过程还需要相关的工具。MTD 工具在这里仍然有效，但是多数情况下不会把这些工具包含到产品中去。多数 Bootloader 具备擦写 Flash 的功能，可以作为软件更新的工具。最原始的办法就是使用硬件仿真器了，系统处理问题还需要它来恢复呢。

下面以 S3C2410 的 U-Boot 为例，说明部署内核和文件系统映像的过程。系统升级过程也完全可以这样做。

```
=> tftp 30100000 uImage
=> erase 40000 13ffff
=> cp.b 30100000 40000 <size>
=> tftp 30100000 initrd.uboot
=> erase 140000 3fffff
=> cp.b 30100000 100000 <size>
```

这样 uImage 写到 Flash 物理地址 0x40000；initrd.uboot 写到 Flash 物理地址 0x140000。使用 U-Boot 的 bootm 命令就可以从本地启动 Linux 系统了。

```
=>bootm 40000 140000
```

当 Linux 从目标板本地启动的时候，一个嵌入式 Linux 产品样机就基本完成了。

“黑色经典”系列之《嵌入式 Linux 系统开发技术详解——基于 ARM》

FARSIGHT

第 14 章 系统设计开发实例

本章目标

本章以 S3C2410 处理器的 GPS 手持设备开发过程为例，介绍了嵌入式 Linux 系统软硬件的设计与开发。通过本章的实例，可以加深对嵌入式 Linux 开发流程的概念的理解，了解嵌入式 Linux 系统开发的基本过程。

- 需求分析
- 系统硬件设计
- 系统软件设计
- 系统集成与部署

14.1 需求分析

随着全球定位系统（GPS，Globe Positioning System）的广泛应用，GPS 手持终端设备的市场需求越来越大。本项目要设计开发一款手持 GPS 工程样机。

首先，分析系统工作原理，再选择合适的参考硬件平台，然后选择合适的操作系统和软件。

（1）系统工作原理

系统首先通过 GPS 模块获得绝对位置数据，并将数据通过 UART 通信方式传给处理器，经处理器处理后得到当前地图的相对位置，并实时显示到 LCD 上，使用户随时知道自己的方位；键盘和触摸屏作为人机接口，进行进一步的查询工作；USB 用于同微机通信，是可选功能；存储器单元用于存储数据，包括操作系统和应用程序。

（2）选择参考硬件平台

目前半导体供应商提供各种 ARM/XSCALE 体系结构的处理器，在第 2 章描述了各种处理器的特点。有些 ARM926 和 XSCALE 核的处理器都已经用来设计手机等高端移动通信设备，有些则可以用来设计中低端的 PDA 设备。考虑价格因素，采用 Samsung 的 S3C2410 ARM920T 处理器。S3C2410 属于中低端的处理器，适合手持设备并且国内外参考硬件平台很多，系统大部分模块在硬件平台上可以直接测试。

系统包括 CPU、GPS 模块、存储器单元、LCD 模块、触摸屏、键盘、USB 接口等部分组成，功能框图如图 14.1 所示。

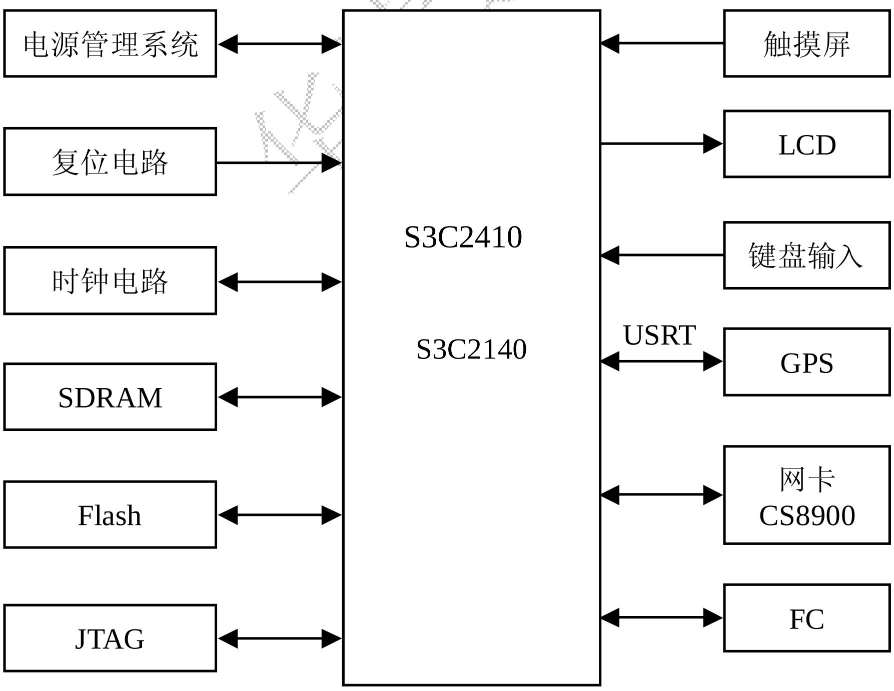


图 14.1　系统功能框图

根据项目的需求，可以适当调整硬件配置。例如：总共需要多少存储空间，使用什么型

号的 Flash 等。同时也要考虑软件支持的程度，驱动程序能否支持新的接口芯片等。本项目的硬件参考配置如下。

- CPU 单元（S3C2410X 16/32-bit ARM920T 内核）
- 存储器单元（2MB NOR Flash 和 32MB NAND Flash，64MB SDRAM）
- 复位电路（包括上电复位和手段复位，至少保持 4 个时钟周期的有效低电平，保证系统的可靠复位）
- 电源电路
- 时钟电路（外部 12MHz 时钟输入，经内部 PLL 倍频至 200MHz 及 32.768KHz 的 RTC 时钟输入）
- 实时时钟（内部 RTC 带日历功能）
- LCD 接口
- 触摸屏接口
- 键盘接口
- GPS 模块
- USB 接口

（3）选择操作系统和软件

Samsung S3C2410 是最开放的开发设计平台。它的软件和硬件设计资料可以免费从互联网上获取，并且 Windows CE 和 Linux 两种操作系统都能支持。在参考板上一般可以直接对 Windows CE 和 Linux 进行测试。

Linux 对 S3C2410 处理器支持得相当好，操作系统内核和应用程序都是开放源码的。因此，Linux 可以完全按照自己的需要裁减配置系统，使得尺寸更小，而且 Linux 操作系统性能比 Windows 操作系统性能更优越。另外，可以避免支付 Windows 产品的版税。S3C2410 在 Linux 社区有庞大的用户群，可以获取丰富的开发调试信息。

我们选择 Linux 作为工程样机的操作系统。由于图形界面要求并不复杂，可以选择 QT/Embedded 图形系统。

14.2 系统硬件设计

1. 电源电路设计

电源电路的设计是非常重要的，特别是对于手持设备来说，如何减少系统的功耗往往成为工程师最为头疼的问题。不过本文不会深入研究如何降低功耗，只是用一个最普通的方案实现。

电源框图如图 14.2 所示。

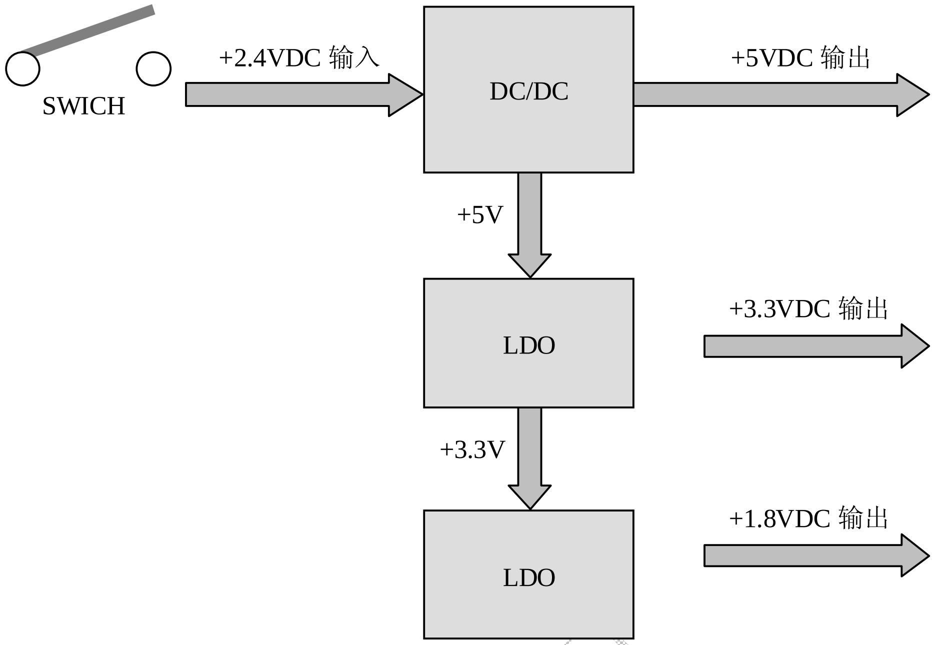


图 14.2　电源管理系统功能框图

电源由电池提供的 2.4V 电压输入，经 DC/DC 开关电源变换器升压至 5V 输出，3.3V 由低压差线性电压源 LDO 通过＋5V 调整输出得到，1.8V 则由 LDO 通过 3.3V 调整输出得到，这样完成了系统供电。我们知道电源分开关电源和线性电源，那么在嵌入式系统中我们该如何选择呢？每种方式都有自己的优点和弱点，表 14.1 总结了这两种电源的优缺点。

表 14.1　　开关电源和线性电源的优缺点

	优　　点	弱　　点
开关电源	输入电压范围宽 效率高 输出功率大 应用比较灵活	电路相对复杂，外围器件较多 对电容电感的要求很高，布线也很讲究 开关频率会给系统带来干扰 纹波比较难控制
线性电源	电路简单，外围器件很少 输出精度高，有很好的负载曲线 工作在低频状态，不会给系统带来麻烦	输入范围比较受限制 效率低，线性电源自身的损耗造成的 输出功率相对较小

线性电源通常由一只工作在电网频率的变压器、一个桥式整流器和一只电容器组成。变压器能够升压也能够降压，同时还与电网隔离。交流正弦波经过桥式整流器整流后的信号，再用电容器平滑为直流电压，这是未经调节的直流电压。为了在输出端得到调节性能较好的电压，增加了一只线性电压调节器。这样，电压的调节性能较好，但是电源的元件数量增多了，成本也提高了，效率通常低于 50%。

开关电源输出电压的调节性能很好，频率较高，可以在输入端加上通用范围的电压，而且发热少、尺寸小、重量轻。开关电源的主要元件有脉冲宽度调制驱动器、MOSFET 功率晶体管、一只变压器以及反馈电路。效率通常高于 50%，既省电，元器件的寿命也更长。

根据以上的分析，在输入输出压降幅度大、功耗高，或是要求升压的场合，往往采用开关电源方案；在压降小、功率要求不是很大的时候，使用线性电源为宜。电源是整个系统中最重要的环节，大多数不稳定的因素或故障都是由于电源方面的设计造成的，因此必须加以重视，否则后患无穷。

本例中使用的 DC/DC 变换器是 ST（意法半导体）公司的 L6920，该产品广泛用于手持系统、移动电话、数模照相机等应用领域，原理图如图 14.3 所示。L6920 管脚说明如表 14.3 所示。

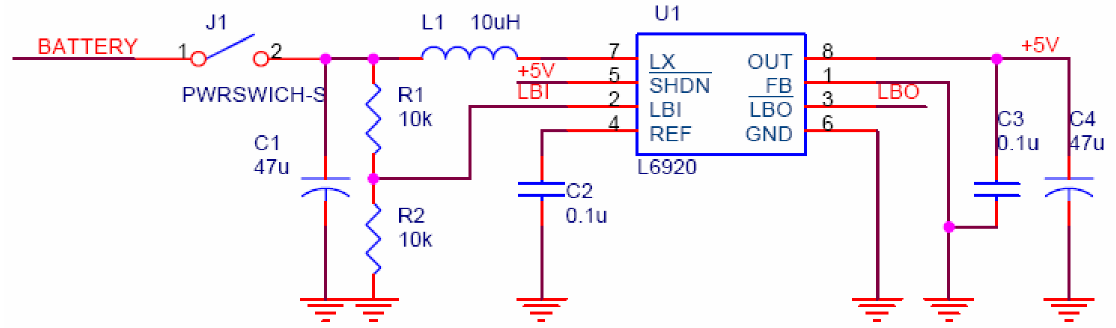


图 14.3 L6920 电路原理图

表 14.2 L6920 管脚说明

引 脚	功 能
FB	电压反馈端，接地输出为 5V，接 Vout 输出为 3.3V，若要求在 2～5.2V 之间，需要并入电阻进行反馈
LBI	电池电压检测输入端 V_{LBI}＝1.23V（1+R1/R2）
LBO	当 V_{LBI} 低于 1.23V 时，输出低电平，通知 CPU 作准备。注意，该管脚是集电极开路输出，需要一个 200k 左右的上拉电阻
REF	1.23V 参考电压，需要在地之间加入 o.1μF 的旁路电容，用于过滤高频噪声
SHDN	断电输入端。当输入电压低于 0.2V 时掉电，高于 0.6V 正常工作
GND	接地
LX	升压电感连接端
OUT	电源输出端

此外，电感电容的选取很重要，直接影响 DC/DC 的效率和输出波形，电感如 COILCRAFTS、COILTRONICS、MURATA 等公司的产品性能非常好，不过价格比较贵。一些我国台湾产的性价比也不错，电容的选取要容易一些。目前大陆生产的钽电容质量都还可以，如果要求不苛刻的话，完全满足要求。线性电源 LDO 的电路就简单得多，这里只列出输出 3.3V 的电路图，如图 14.4 所示。

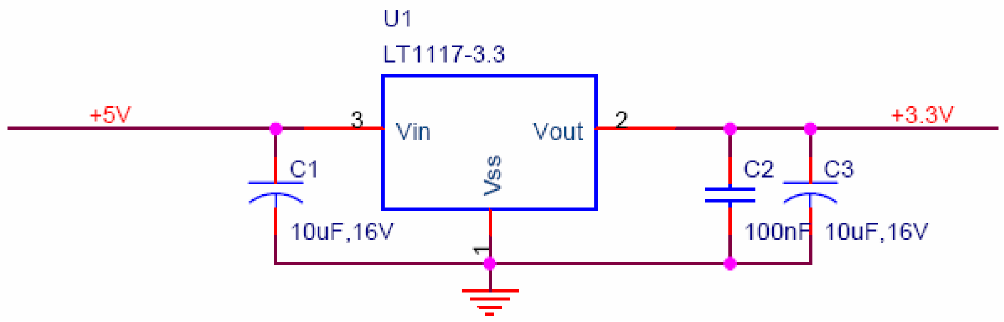


图 14.4 线性电源 LDO 的电路原理图

当今有很多优秀的电源管理系统设计厂商，如 Linear、TI、ADI、ON-SEMI、ST、Semtech，Intersil、Maxim 等，有很大的选择余地。

2．复位电路设计

复位对于一个系统来说很重要，由于各个单元要进入正常工作状态，都需要可靠的复位。正常情况下，一般有上电复位和手动复位。如果电源电压出现波动，系统会非正常复位，这时候会发生复位时间不够从而造成一些错误甚至死机，所以复位监控电路也是必要的复位。电路原理如图 14.5 所示。

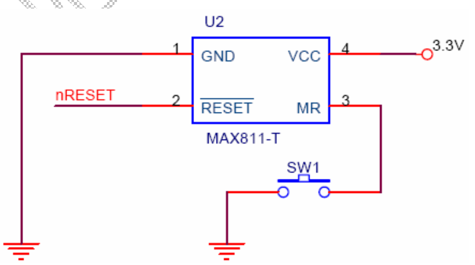


图 14.5 复位电路原理图

我们的复位电路使用的芯片是 MAX811，管脚说明如表 14.3 所示。

根据其型号的后缀，MAX811 的复位电平门限有 5 种规格（表 14.4），例如表中用的是 MAX811-T，就表示电压低于 3.08V 时，产生复位信号。

表 14.3 MAX811 管脚说明

引　脚	功　能
GND	接地
RESET	复位输出端。当系统上电、手动复位、电压波动低于标称值的时候会输出最少 140ms 的低电平脉冲
MR	手动复位输入端。当本管脚接地时，RESET 端产生复位信号
VCC	电源输入

如系统需要中有与 CPU 逻辑相反的复位信号，则在 nRESET 信号后加入非门反相输出。

表 14.4 MAX811 的规格

后　缀	电　压	后　缀	电　压
L	4.63	S	2.93
M	4.38	R	2.63
T	3.08		

3．时钟电路设计

CPU 部分需要两路时钟输入，一路是 CPU 工作时钟输入，另一路提供给 RTC 电路。CPU 工作时钟是一个有源晶振，无需外部电容，直接输出 12MHz 时钟信号到 CPU，由 CPU 内部 PLL 倍频到 200MHz，两路时钟输入如图 14.6 所示。这 2 种是最普遍的时钟电路，在后面的 CS8900A 接口电路中也有时钟电路，做法是一样的。

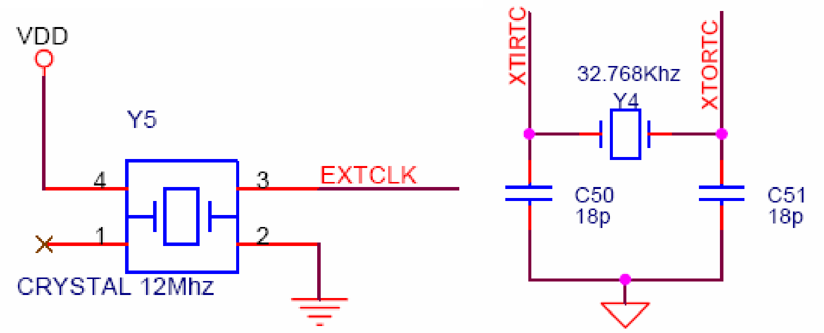


图 14.6　时钟电路原理图

4．SDRAM 接口电路设计

SDRAM 具有容量大，存取速度快，成本低的特点，因而广泛应用到微机处理系统中。SDRAM 主要用来存放执行代码和变量，是系统启动之后主要进行存取操作的存储器。由于 SDRAM 需要定时刷新以保持存储的数据，因而要求微处理器具有刷新控制逻辑，或在系统中另外加入刷新控制逻辑电路。S3C2410X 及其他一些 ARM 芯片在片内具有独立的 SDRAM 刷新控制逻辑，可方便的与 SDRAM 接口。但某些 ARM 芯片则没有 SDRAM 刷新控制逻辑，就不能直接与 SDRAM 接口，在进行系统设计时应注意这一点。

目前常用的 SDRAM 为 8 位/16 位的数据宽度，我们可根据系统需求，构建 16 位或 32 位的 SDRAM 存储器系统。本例中使用的是两片三星 K4S561632C-TC75 芯片构建 32 位的 SDRAM 存储器系统。每片 K4S561632C 的存储容量为 16 组×16M 位（32M 字节），工作电压为 3.3V，常见封装为 54 脚 TSOP，兼容 LVTTL 接口，支持自动刷新（Auto-Refresh）和自刷新（Self-Refresh），16 位数据宽度。其管脚说明如表 14.5 所示。

表 14.5　　K4S561632C-TC75 管脚说明

引　脚		功　能
CLK	时钟	芯片时钟输入
CKE	时钟使能	片内时钟信号控制
nSCS	片选	禁止或使能除 CLK、CKE 和 DQM 外的所有输入信号
BA0，BA1	组地址选择	用于片内 4 个组的选择
A12～A0	地址总线	行地址：A12～A0，列地址：CA8～CA0，自动预充电标志：A10
NSRAS nSCAS nWE	行地址锁存 列地址锁存 写使能	参照功能真值表，nRAS，nCAS 和 nWE 定义相应的操作
LDQM，UDQM	数据 I/O 屏蔽	在读模式下控制输出缓冲；在写模式下屏蔽输入数据

DQ15～DQ0	数据总线	数据输入输出引脚
VDD/VSS	电源/地	内部电路及输入缓冲电源/地
VDDQ/VSSQ	电源/地	输出缓冲电源/地

SDRAM 连接原理图如图 14.7 所示。

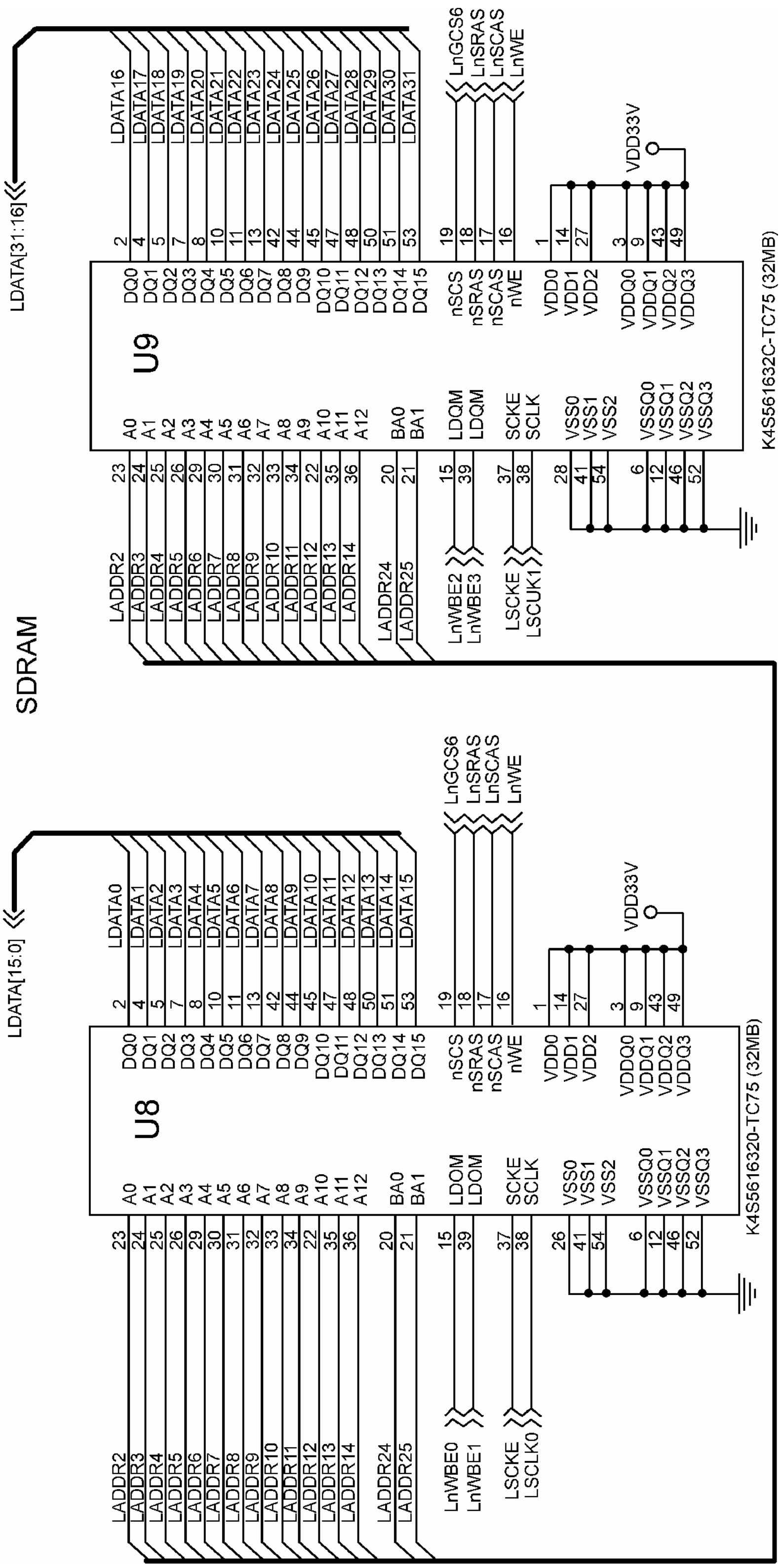


图 14.7 SDRAM 电路原理图

5．Flash 接口电路设计

Flash 存储器是一种在系统上（In-System）进行电擦写，掉电后信息不丢失的存储器。它具有低功耗、大容量、擦写速度快、可整片或分扇区在系统编程（烧写）、擦除等特点，并且可由内部嵌入的算法完成对芯片的操作，因而在各种嵌入式系统中得到了广泛的应用。作为一种非易失性存储器，Flash 在系统中通常用于存放程序代码、常量表以及一些在系统掉电后需要保存的用户数据等。

现在市场上两种主要的非易失闪存技术是 NOR 和 NAND。Intel 于 1988 年首先开发出 NOR Flash 技术，彻底改变了原先由 EPROM 和 EEPROM 一统天下的局面。1989 年东芝公司发表了 NAND Flash 结构，强调降低每比特的成本，更高的性能，并且像磁盘一样可以通过接口轻松升级。

NOR 的特点是芯片内执行，这样应用程序可以直接在 Flash 内运行，不必再把代码读到系统 RAM 中。NOR 的传输效率很高，在 1～4MB 的小容量时具有很高的成本效益，但是很低的写入和擦除速度大大影响了它的性能。

NAND 结构能提供极高的单元密度，可以达到高存储密度，并且写入和擦除的速度也很快。应用 NAND 的困难在于 Flash 的管理和需要特殊的系统接口。

Flash 闪存是非易失存储器，可以对称为块的存储器单元块进行擦写和再编程。任何 Flash 器件的写入操作只能在空或已擦除的单元内进行，所以大多数情况下，在进行写入操作之前必须先执行擦除。NAND 器件执行擦除操作是十分简单的，而 NOR 则要求在进行擦除前先要将目标块内所有的位都写为 0。两者的速度差异如下。

- NOR 的读速度比 NAND 稍快一些。
- NAND 的写入速度比 NOR 快很多。
- NAND 的 4ms 擦除速度远比 NOR 的 5s 快。
- 大多数写入操作需要先进行擦除操作。
- NAND 的擦除单元更小，相应的擦除电路更少。

基于以上分析，我们用 NOR Flash 存储 Boot 代码，NAND Flash 存储操作系统和应用程序代码。前者我们选用 ST 公司的 M29W160（1Mb×16），后者用三星的 K9F5608B（32MB×8）。

M29W160 的单片存储容量为 16M 位（2MB），工作电压为 2.7V～3.6V，采用 48 脚 TSOP 封装或 48 脚 FBGA 封装，16 位数据宽度，可以以 8 位（字节模式）或 16 位（字模式）数据宽度的方式工作。M29W160 仅需单 3V 电压即可完成在系统的编程与擦除操作，通过对其内部的命令寄存器写入标准的命令序列，可对 Flash 进行编程（烧写）、整片擦除、按扇区擦除以及其他操作。此外，它还支持任意地址的块写保护，十分方便用户。M29W160 的管脚说明如表 14.6 表示。

表 14.6　　M29W160 的管脚说明

引　脚	类　型	功　能
A[19:0]	I	地址总线。在字节模式下，DQ[15]/A[-1]用作 21 位字节地址的最低位
DQ[15]/A[-1] DQ[14:0]	I/O 三态	数据总线。在读写操作时提供 8 位或 16 位的数据宽度。在字节模式下，

		DQ[15]/A[-1]用作 21 位字节地址的最低位，而 DQ[14:8]处于高阻状态

续表

引　　脚	类　　型	功　　能
BYTE	I	模式选择。低电平选择字节模式，高电平选择字模式
E	I	片选信号，低电平有效。进行读写操作时，该引脚必须为低电平，当为高电平时，芯片处于高阻旁路状态
G	I	输出使能，低电平有效。在读操作时有效，写操作时无效
W	I	写使能，低电平有效。在进行编程和擦除操作时，控制相应的写命令
RESET	I	硬件复位，低电平有效。当复位时，立即终止正在进行的操作
RB	O	就绪/忙状态指示。用于指示写或擦除操作是否完成。在进行编程或擦除操作时，该引脚位低电平，操作完成时为高电平，此时可读取内部的数据
VCC	--	3.3V 电源
VSS	--	接地

M29W160 的实际应用原理图如图 14.8 所示。

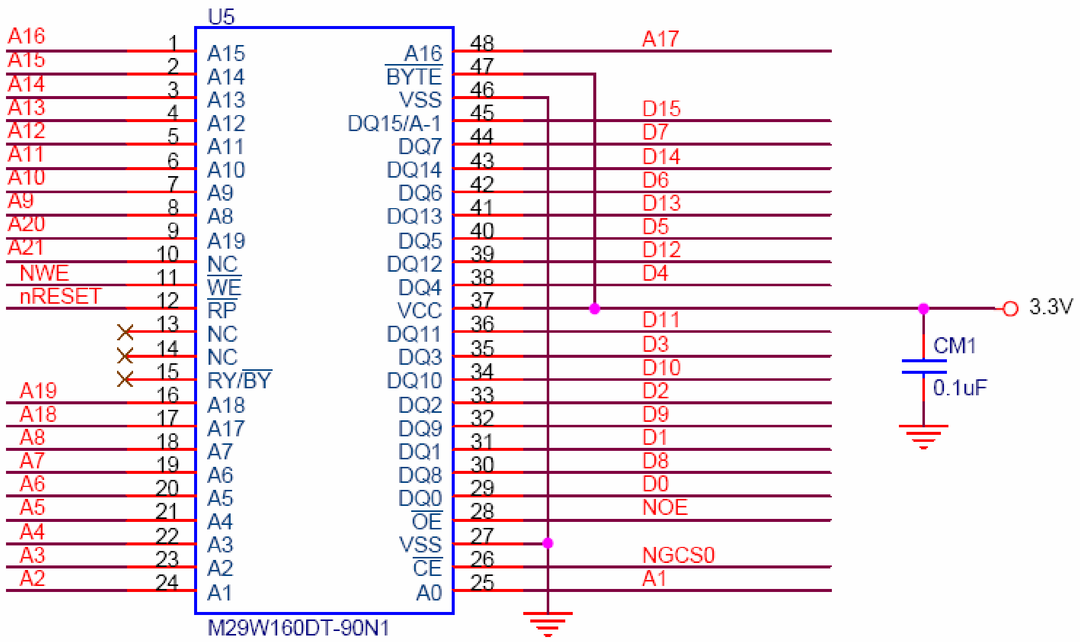


图 14.8　M29W160 电路原理图

三星公司的 K9F 系列 NAND Flash 存储器有着容量大、功耗低、成本低、电气性能好等特点，是大容量 Flash 市场的有力竞争者。K9F5608B 的单片存储容量为 256Mbit，工作电压为 2.7V～3.6V，采用 48 脚 TSOP 封装或 63 脚 TBGA 封装，8 位数据宽度，带有硬件数据保护功能，支持上电自动引导功能，块擦写时间为 2ms，数据存储有效时间达 10 年以上。K9F5608B 的管脚说明如表 14.7 所示。

表 14.7　　　　　　　　K9F5608B 的管脚说明

引　　脚	类　　型	功　　能
I/O0-7	I/O	数据输入输出，地址输入，命令输入
AL	I	地址锁存使能
CL	I	命令锁存使能
E	I	片选信号，低电平有效
R	I	读使能，低电平有效
W	I	写使能，低电平有效
WP	I	写保护，低电平有效
RB	O	就绪/忙状态指示。用于指示写或擦除操作是否完成。在进行编程或擦除操作时，该引脚位低电平，操作完成时为高电平，此时可读取内部的数据
VCC	--	3.3V 电源
VSS	--	接地

K9F5608B 的实际应用原理图如图 14.9 所示。

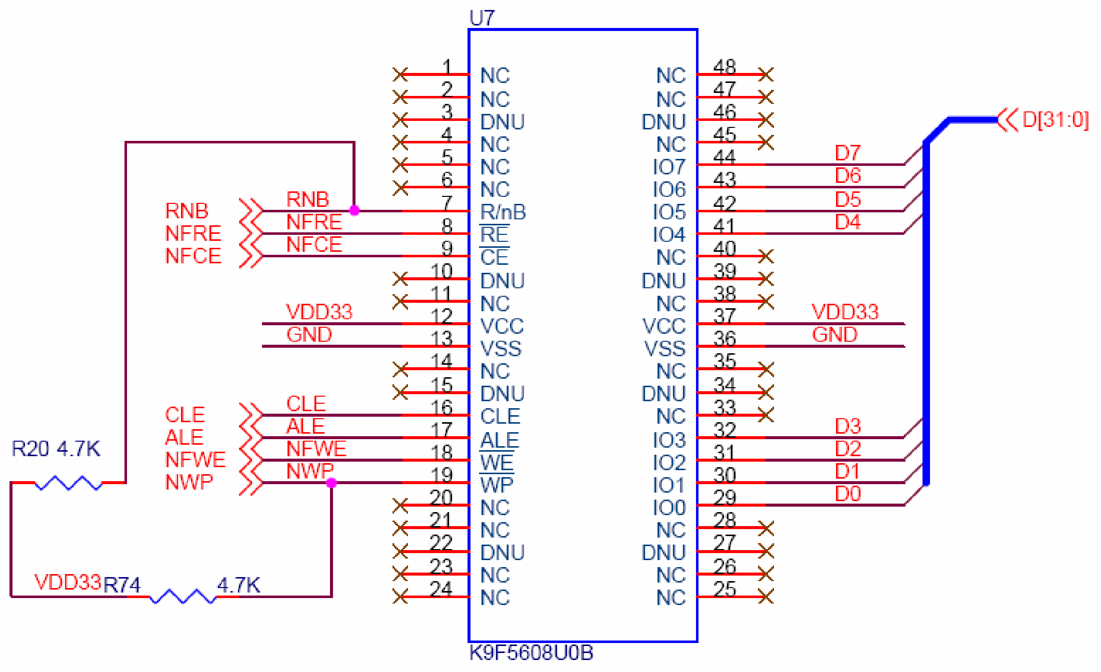


图 14.9　K9F5608B 电路原理图

随着嵌入式技术的高速发展，Flash 被广泛应用到各个领域，功能越来越强大，电路也日趋复杂。不过各大存储器生产厂商采用统一的标准进行设计生产，所以常用的型号往往是完

全兼容的，可以直接替代。我们在应用 Flash 时，只要注意遵循数据手册的指导，按说明进行设计即可，一般不会出现问题。

6．JTAG 接口电路设计

JTAG 接口对于开发调试非常重要。对于产品来说也可以通过 JTAG 修复或者更新软件，所以最好预留这个接口，至少引出 JTAG 接口线。JTAG 接口电路原理图如图 14.10 所示。

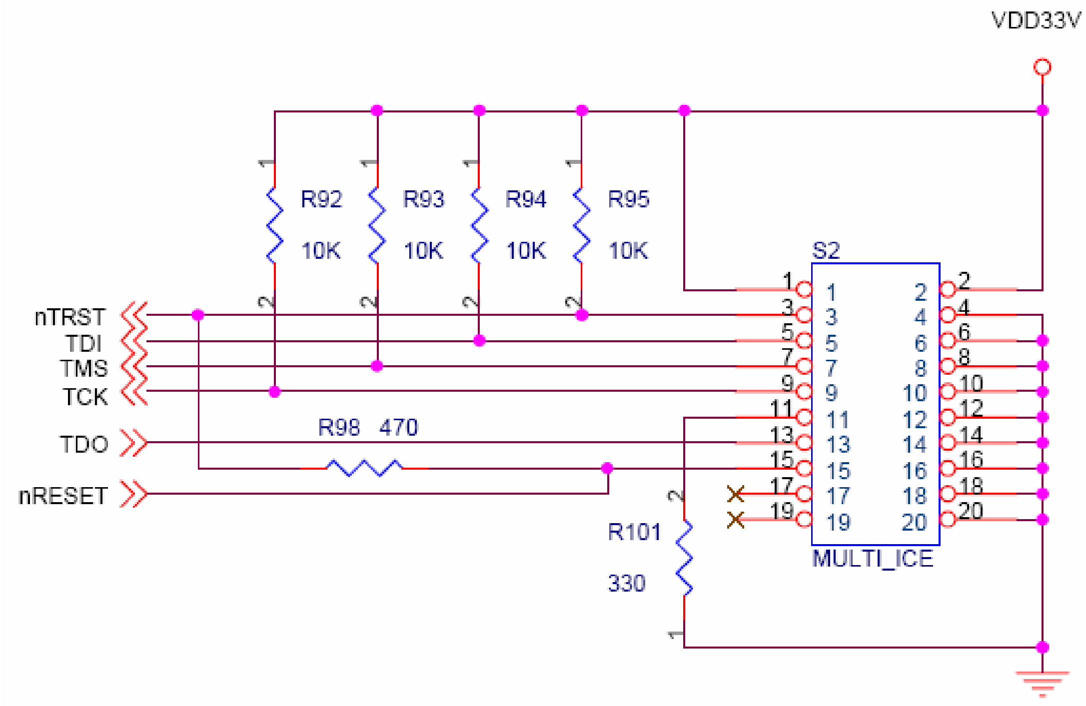


图 14.10　JTAG 电路原理图

7．USB 接口电路设计

S3C2410X 提供了方便的 USB 1.1 接口，片内包括 2 个 USB 控制器，可设置为两个主机或 1 主机 1 设备。

USB 接口原理图如图 14.11 所示。

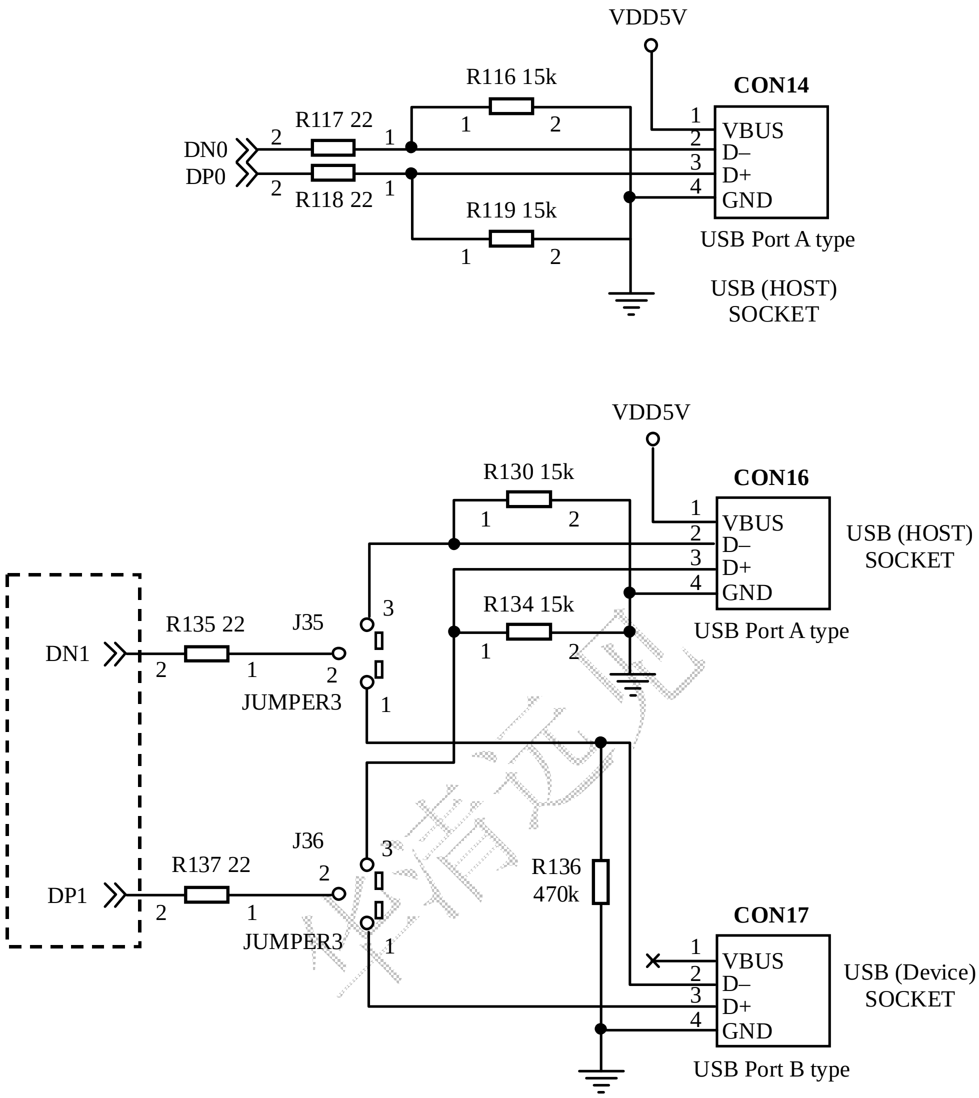


图 14.11 USB 电路原理图

接口说明：CON14 工作在 A-TYPE（主机）状态，跳线 J35、J36 的 2、3 脚接通时 CON16 有效，工作在 A-TYPE（主机模式）状态，跳线 J35、J36 的 1、2 脚接通时，CON17 有效，工作在 B-TYPE（设备模式）状态。

8．键盘输入接口电路设计

键盘采用中断方式连接，共 6 个功能键，占用 6 个中断源。当有按键被按下时，会实时产生中断请求信号，通知 CPU 处理。电路比较简单，如图 14.12 所示。

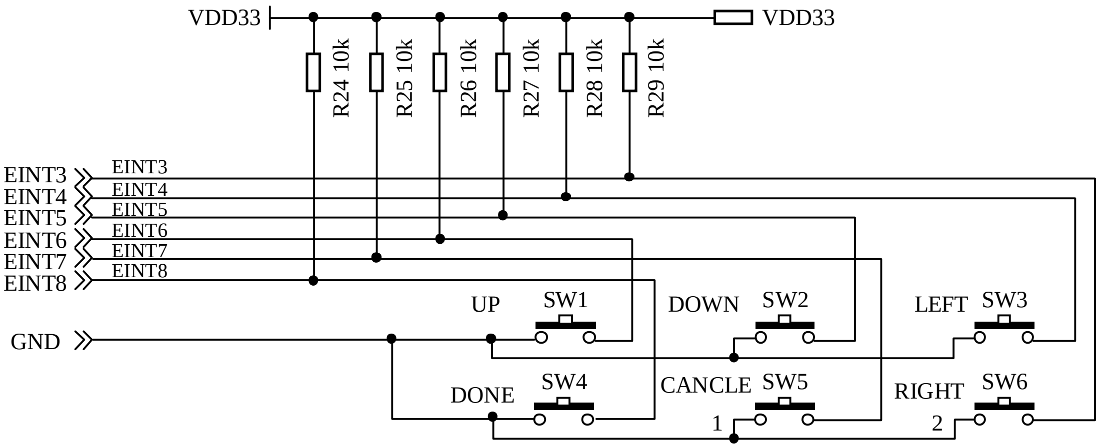


图 14.12　键盘输入电路原理图

9．LCD（触摸屏）接口电路设计

S3C2410X 支持 TFT/STN 型的 LCD 及触摸屏，但是不能直接与 LCD 相连，需要接口板驱动 LCD。S3C2410X 通过 50pin 的插座作为 LCD 与触摸屏接口，至于接口板，市场上可以买到，管脚定义都是标准的。

原理图如图 14.13 所示。

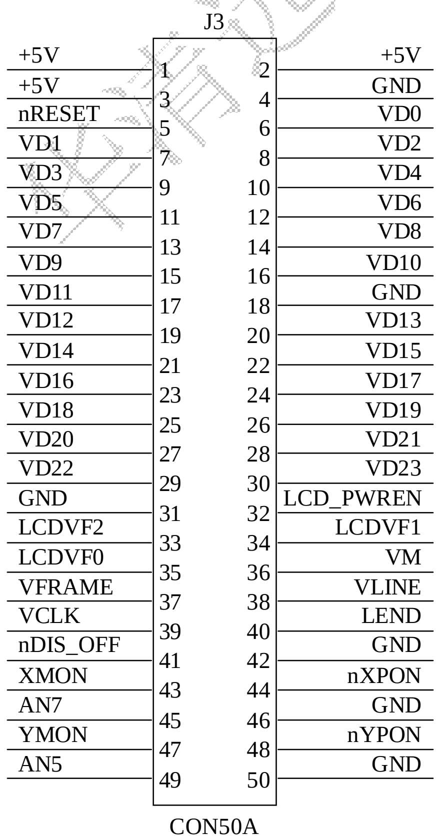


图 14.13　LCD 电路原理图

10．GPS 接口电路设计

本设计选用的 GPS 模块是 HIMARK 的 AR2010-GM，该产品体积小、功耗低，具备标准 GPS 功能，技术参数如下。

定位精度 5～25 m CEP w/o S/A；

速率 0.1 m/sw/o S/A；

时间±1μsec（卫星时间）；

冷起动<90 s；

暖起动<45 s；

热起动<15 s；

海拔高度 max.18000 m；

速度 max.500 m/s ；

加速度 max.±4g；

导航修正速率 1/s；

端口 RS232 串口 x1，TTL x1；

速率 4.8、9.6、19.2 和 38.4kbit/s（可选择升级至 115.2kbit/s）；

输出协议 NMEA 0183：GGA、GLL、GSV、GSA、RMC、VTG；

电源 3 VDC～9VDC；

消耗电流小于 27mA；

卫星信道 12-channel；

工作温度–40℃～85℃；

储存温度–55℃～100℃；

工作适度 5%～95%；

AR2010-GM 模块通信端口定义如表 14.8 所示。

表 14.8　　AR2010-GM 模块通信端口定义

引　　脚	功　　能	备　　注
VCC	电源输入	
TXD-232	数据发送端，RS232 电平	数据传输率可设置为 4800/9600/19200/38400，最高至 115200bit/s，8 位数据，1 停止位，无奇偶校验
RXD-232	数据接收端，RS232 电平	2 进制数据输入
RXD-TTL	数据接收端，TTL 电平	2 进制数据输入
GND	地线	
TXD-TTL	数据发送端，TTL 电平	数据传输率可设置为 4800/9600/19200/38400，最高至 115200bit/s，8 位数据，1 停止位，无奇偶校验

虽然 AR2010-GM 模块的功能强大，内部结构复杂，但是与 CPU 的通信非常简单，采用标

准 UART 方式，只需 4 根信号线即可，无需任何附加元件就能和 S3C2410X 连接。

11．PCB 设计制作

随着处理器频率的不断加快，对 PCB 的布局和走线要求越来越高，早已不是仅仅布通就可以的时代了。当今的 PCB 板制造技术向高密度、高精度、细孔径、细导线、细间距、高可靠、多层化、高速传输、轻量、薄型方向发展，在生产上同时向提高生产率、降低成本、减少污染、适应多品种、小批量生产的方向发展。

对于主频高达 200MHz 的 S3C2410X 处理器来说，它的 PCB 设计是硬件工作中的难点，前面所有工作都集中体现在电路板上，因而 PCB 的设计直接影响整个系统的性能。掌握好 PCB 设计的步骤和要点，不仅能加快开发进度，更会增加系统的稳定性，也为软件开发创造更好的条件。在 PCB 布板之前，一定要保证原理图的正确性，PCB 的电气关系是由原理图的网表生成的，错误的原理图会直接导致错误的 PCB 板，还要重新返工，这一点相信所有的硬件工程师都有切身体会。

14.3 系统软件设计

在样机硬件设计阶段，就可以在参考硬件平台上进行软件开发。

软件开发是一个反复修改编译的过程，交叉开发环境的软件开发流程如图 14.14 所示。

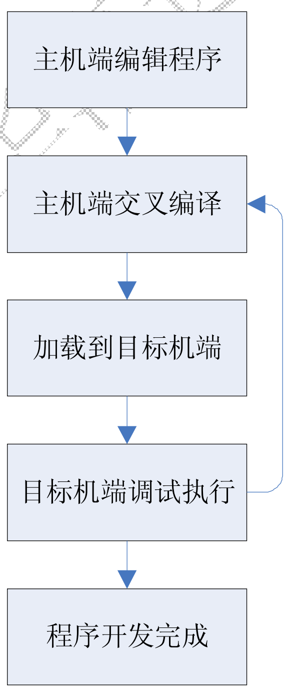


图 14.14 程序开发流程

当样机电路板做好以后，还需要移植到新的平台上。因为样机是参照参考板设计的，所以大部分接口（特别是片上的设备接口）都相同。但是可能增减个别的接口或者替换外围芯片，例如：Flash 由 AMD 公司的 Am29LV800B 替换成 ST 公司的 M29W160（1Mbit×16），容量增加了，擦写 Flash 的指令也要相应修改。

软件移植包括 Linux 系统的 3 个组成部分：U-Boot 移植、内核移植和应用程序移植。样机开发板名称假设为 GPS2410，程序移植时将引用这个名称。

1．U-Boot 的移植

S3C2410 的参考平台上一般使用 vivi 作为 Linux 引导程序。如果不习惯 vivi 或者觉得 U-Boot 好用，那就把 U-Boot 移植到硬件板上吧。第 6 章详细介绍了 U-Boot 的开发使用，对于 U-Boot 的移植是很有帮助的。

（1）在顶层 Makefile 中为开发板添加新的配置选项，添加下面 2 行：

```
gps2410_config   :      unconfig
         @./mkconfig $(@:_config=) arm arm920t gps2410 NULL s3c24x0
```

（2）创建一个新目录存放开发板相关的代码，并且添加文件：

```
board/gps2410/config.mk
board/gps2410/flash.c
board/gps2410/gps2410.c
board/gps2410/Makefile
board/gps2410/memsetup.S
board/gps2410/u-boot.lds
```

这些文件都可以先从 smdk2410 目录下复制过来，再把名称改一下。

（3）为开发板添加新的配置文件

可以先复制参考开发板的配置文件，再修改。例如：

```
$ cp include/configs/smdk2410.h  include/configs/gps2410.h
```

（4）配置开发板

```
$ make gps2410_config
```

（5）编译 U-Boot

执行 make 命令，编译成功可以得到 U-Boot 映像。有些错误是跟配置选项是有关系的，通常打开某些功能选项会带来一些错误。一开始可以尽量跟参考板配置相同。

（6）添加驱动或者功能选项

GPS2410 开发板的以太网驱动仍然采用常见的 CS8900 以太网接口芯片。可能变化的是以太网控制寄存器基地址，只要在 include/config/gps2410.h 文件中修改即可。非常幸运，只要改一下基地址，gps2410 的以太网接口就能够正常工作了。

因为 Flash 换成了 ST 公司的 M29W160，所以必须修改 Flash 的擦写函数。阅读 Flash 芯片手册或者擦写例程，确定 Flash 的扇区大小、块大小、块数和芯片大小，明确 Flash 的擦写操作指令。然后在源程序 board/gps2410/flash.c 中修改。

（7）调试 U-Boot 源代码，直到 U-Boot 在开发板上能够正常启动。

这最好借助硬件仿真器来调试，至少需要烧写 Flash 的工具。

2．Linux 内核的移植

S3C2410 属于片上系统，处理器芯片具备串口、显示等外围接口的控制器。参考板上的设备驱动程序多数可以直接使用。但是有些驱动程序也需要移植或者重新开发。

Linux-2.6.14 的内核可以对 S3C2410 有基本的支持。

第 1 个驱动是串口控制台。2.6 内核的串口设备名称有了改变，由“ttyS”变成了“ttySAC”。所以只要内核命令行参数相应的修改成为：console=ttySAC0,115200。

第 2 个驱动是以太网接口。GPS2410 开发板的网络接口使用 CS8900A 芯片，但是 Linux 2.6 内核中并没有支持 S3C2410 平台的驱动。幸运的是从 Linux 社区搜到了针对 CS8900 接口的补丁，这可省了不少力气。不然，还要认真分析一下网络驱动程序了。

第 3 个驱动是 LCD 驱动。这个驱动程序是 s3c2410 的 Frame Buffer 驱动：s3c2410fb，源文件是 drivers/video/s3c2410fb.c。问题是参考板和样机的 LCD 不是同一品牌的，有些显示参数的设置需要修改。

其他的驱动跟硬件参考平台很接近，只需要小修小改，就可以跑通了。

最后还要考虑一下 MTD 驱动。MTD 可以对 Flash 建立分区表，以便分别存储 U-Boot、内核和文件系统。

3．应用程序的开发移植

应用程序的功能包含 2 个方面：一方面是实现图形用户界面，另一方面是 GPS 数据处理。

图形用户界面可以基于 QT/Embedded 实现。使用 QT/E 空间可以开发一些简单便捷的图形应用程序，可以支持触摸屏和按键接口。

GPS 数据是通过串口通信的，所以这部分程序主要是 Linux 串口编程。另外程序还需要具体分析处理接收和发送的数据。

当然也离不开 Linux 基本的系统工具，使用 Busybox 工具集就可以了。

14.4　系统集成与部署

各部分软件开发完成以后，就可以对样机进行彻底测试了。可以先通过 NFS 方式测试，再部署到 Flash 中测试。当然，这很可能是反复测试的过程。对于 GPS2410 样机，只对功能方面做了测试，没有测试性能方面的问题。

本项目采用了 RAMDISK 文件系统，因为它是简单实用的。为了尽量使文件系统最小，按照下列步骤定制文件系统。

（1）创建目录结构

在工作区中建立新目录 myfs，作为定制文件系统。执行下列命令创建目录结构。

```
$ cd ~/workspace
```

```
$ mkdir ./myfs
$ cd ./myfs
$ mkdir bin dev etc lib mnt proc sbin usr
```

（2）创建设备节点

如果内核配置了 devfs，就不需要创建这些节点了。如果找到一个 MAKEDEV 脚本，可以很快地创建所有节点。

```
$ cd dev/
$ mknod –m 660 mtd0 c 90 0
$ mknod –m 660 mtd1 c 90 2
$ mknod –m 660 mtd2 c 90 4
$ mknod –m 660 mtdblock0 b 31 0
$ mknod –m 660 mtdblock1 b 31 1
$ mknod –m 660 mtdblock2 b 31 2
```

（3）添加应用程序

编译安装 Busybox 及其链接，可以直接从测试过文件系统中复制。

```
$ cd ~/workspace/myfs
$ cd ./bin
$ cp /usr/local/arm/3.3.2/rootfs/bin/busybox ./
$ ln -s /bin/busybox basename
$ ln -s /bin/busybox cat
$ ln -s /bin/busybox chgrp
$ ln -s /bin/busybox chmod
$ ln -s /bin/busybox chown
$ ln -s /bin/busybox clear
$ ln -s /bin/busybox cp
$ ln -s /bin/busybox cut
$ ln -s /bin/busybox date
$ ln -s /bin/busybox dd
$ ln -s /bin/busybox df
$ ln -s /bin/busybox dmesg
$ ln -s /bin/busybox du
$ ln -s /bin/busybox echo
$ ln -s /bin/busybox env
$ ln -s /bin/busybox expr
$ ln -s /bin/busybox false
$ ln -s /bin/busybox free
$ ln -s /bin/busybox grep
```

```
$ ln -s /bin/busybox gunzip
$ ln -s /bin/busybox gzip
$ ln -s /bin/busybox head
$ ln -s /bin/busybox hostname
$ ln -s /bin/busybox kill
$ ln -s /bin/busybox killall
$ ln -s /bin/busybox ln
$ ln -s /bin/busybox ls
$ ln -s /bin/busybox mkdir
$ ln -s /bin/busybox mknod
$ ln -s /bin/busybox more
$ ln -s /bin/busybox mount
$ ln -s /bin/busybox mv
$ ln -s /bin/busybox ping
$ ln -s /bin/busybox ps
$ ln -s /bin/busybox pwd
$ ln -s /bin/busybox rdate
$ ln -s /bin/busybox rm
$ ln -s /bin/busybox rmdir
$ ln -s /bin/busybox sed
$ ln -s /bin/busybox sh
$ ln -s /bin/busybox sleep
$ ln -s /bin/busybox sync
$ ln -s /bin/busybox tail
$ ln -s /bin/busybox telnet
$ ln -s /bin/busybox test
$ ln -s /bin/busybox time
$ ln -s /bin/busybox true
$ ln -s /bin/busybox umount
$ ln -s /bin/busybox uname
$ ln -s /bin/busybox uptime
$ ln -s /bin/busybox usleep
$ ln -s /bin/busybox vi
$ ln -s /bin/busybox wget
$ ln -s /bin/busybox zcat
$ cd ../sbin
$ ln -s /bin/busybox halt
$ ln -s /bin/busybox ifconfig
$ ln -s /bin/busybox init
```

```
$ ln -s /bin/busybox reboot
$ ln -s /bin/busybox route
```

另外还要把开发的图形和应用程序安装到相应目录下。

（4）添加库

应用程序运行是依赖于动态库的，还需要安装依赖的库。

```
$ cd lib
$ cp /opt/target/rootfs/lib/ld-linux.so.2 ./
$ cp /opt/target/rootfs/lib/libc.so.6 ./
$ cp /opt/target/rootfs/lib/libnss_dns.so.2 ./
$ cp /opt/target/rootfs/lib/libnss_files.so.2 ./
$ cp /opt/target/rootfs/lib/libresolv.so.2 ./
```

（5）添加配置脚本

Linux 系统的启动脚本大部分在 etc 目录下，复制 etc 下的一些脚本，并且配置修改。也可以自己定制一个简单的脚本/linuxrc，完成全部初始化工作。

```
#/linuxrc
```

（6）制作 RAMDISK

```
$ mkdir /mnt/initrd
$ dd if=/dev/zero of=initrd.img bs=1k count=8192
$ mkfs.ext2 –F initrd.img
$ mount –o loop initrd.img /mnt/initrd
$ cp –a ./myfs/* /mnt/initrd
$ umount /mnt/initrd
$ gzip –best –c initrd.img > initrd.img.gz
```

initrd.img.gz 就是一个 RAMDISK 文件系统映像的压缩格式。

接下来还要重新配置编译一下内核。去掉一些用于交叉开发的选项，还要修改内核命令行参数，确定启动挂接本地的 RAMDISK 文件系统。

因为使用 U-Boot 引导程序，需要使用 U-Boot 格式的内核映像和 RAMDISK 映像。内核映像只要执行“make uImage”就可以得到。RAMDISK 需要通过 mkimage 工具手工转换，命令如下。

```
$ mkimage -n 'RAM disk' -A arm -O linux -T ramdisk -C gzip –a 0x30800000 –e 0x30800000 -d initrd.img.gz initrd.uboot
```

然后，通过 U-Boot 把 uImage 和 initrd.uboot 烧写到 Flash 对应分区中去。

```
=> tftp 30100000 uImage
=> erase 40000 13ffff
=> cp.b 30100000 40000 100000
=> tftp 30100000 initrd.img.gz
=> erase 140000 3fffff
```

```
=> cp.b 30100000 100000 200000
```

最后，还要设置 U-Boot 环境变量，让系统自动启动。

```
=> setenv bootargs root=/dev/ram rw initrd=0x30800000,8M
=> setenv bootcmd bootm 40000 140000
=> setenv bootdelay 3
=> saveenv
```

系统复位，Linux 系统就可以完全自动地从本地 Flash 中启动了。

至此，一个 GPS 定位系统可以正常工作了。但是考虑到产品成本控制，升级和维护，真正产品化还需要大量工作。